Prelude

A typical dissertation will be structured according to (somewhat) standard sections, described in what follows. However, it is hard and perhaps even counter-productive to generalise: the goal is *not* to be prescriptive, but simply to act as a guideline. In particular, each page count given is important but *not* absolute: their aim is simply to highlight that a clear, concise description is better than a rambling alternative that makes it hard to separate important content and facts from trivia.

You can use this document as a LaTeX-based [**?**, **?**] template for your own dissertation by simply deleting extraneous sections and content; keep in mind that the associated `Makefile` could be of use, in particular because it automatically executes to deal with the associated bibliography.

You can, on the other hand, opt *not* to use this template; this is a perfectly acceptable approach. Note that a standard cover and declaration of authorship may still be produced online via

`http://www.cs.bris.ac.uk/Teaching/Resources/cover.html`

Dissertation Type: research

DEPARTMENT OF COMPUTER SCIENCE

How effective is Deep Reinforcement learning for reducing image noise in Monte Carlo path tracing, compared to Temporal Difference learning?

Callum Pearce

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Engineering in the Faculty of Engineering.

Friday 3rd May, 2019

Declaration

This dissertation is submitted to the University of Bristol in accordance with the requirements of the degree of MEng in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

Callum Pearce, Friday 3rd May, 2019

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

Executive Summary

In the field of Computer Graphics, Path tracing is an algorithm which accurately approximates global illumination in order to produce photo-realistic images. Path tracing has traditionally been known to trade speed for image quality. This is due to the lengthy process of accurately finding each pixels colour, whereby many light rays are fired through each pixel into scene, then directions for each ray are continually sampled until it intersects with a light source. Due to this, a variety of Importance sampling algorithms have been designed to avoid sampling directions which lead to rays contributing no light to the rendered image. The paths formed by sampling rays in these directions are known as zero contribution light paths. By not sampling zero contribution light paths, it is possible to significantly reduce the noise in rendered images using the same number of sampled rays per pixel in path tracing

Recently a Temporal Difference learning method was used by Nvidia to achieve impressive results in Importance sampling within a Path tracer. The algorithm essentially learns which directions light is coming from for a given point in the scene. It then uses importance sampling to favour shooting rays stored in those directions, reducing the number of zero contribution light paths sampled. With this success, there is plenty of potential to experiment with other Temporal Difference learning methods, particularly Deep Q-Learning. It is also important to assess both of these methods on their ability to accurately approximate Global Illumination to produce photo-realistic images. From this, my goal is to investigate the ability of two different temporal difference learning algorithms ability to reduce the number of zero contribution light paths in path tracing, whilst still accurately approximating global illumination. More specifically, the first temporal difference learning method will be that proposed by Nvidia, and the second will be my designed Neural-Q path tracing algorithm. I will be comparing these two methods in order to test the following hypothesis:

> The Neural-Q path tracer is further able to reduce the number of zero contribution light paths than an Expected SARSA Path tracer proposed by Nvidia, whilst still accurately simulating Global Illumination.

Outcomes

- Which is better able to reduce the number of zero contribution light paths expected SARSA or Deep Q-learning
- Can Expected SARSA learning handle multiple lights well in a scene & deep q-learning

Main areas of work

- I have written x lines of code to build a Path tracing engine from scratch which supports a variety of GPU accelerated Path tracing algorithms I have experimented with.
- I have spent x hours researching into the field of efficient light transport simulation for ray-tracing techniques.
- I have spent x hours researching into Reinforcement learning, particularly Temporal Difference learning and Deep Reinforcement learning, neither of which I have been taught before.
- I spent x hours implementing and validating the on-line Expected SARSA Path tracing algorithm proposed by Nvidia, which required me to implement the Irradiance Volume data structure as a prerequisite.
- I have spent x hours designing, implementing and analysing my own on-line Deep Q-learning Path tracing algorithm, along with a neural network architecture designed for the algorithm.

Supporting Technologies

1. I used the `SDL2` library for displaying and saving rendered images from my Path tracing engine.
2. I used the `OpenGL` mathematics library to support low level operations in my Path tracing engine. It includes GPU accelerated implementations for all of its functions.
3. I used the `CUDA Toolkit 10.1` parallel computing platform for accelerating Path tracing algorithms. This means the `CUDA nvcc` compiler must be used to compile my Path tracing engine.
4. All experiments were run on my own desktop machine with an Nvidia `1070Ti` GPU, Intel `i5-8600K` CPU and 16GB of RAM.
5. I used the C++ API for the `Dynet` neural network framework to implement all of my Neural Network code as it is able to be compiled by the `CUDA` compiler.

Notation and Acronyms

TD learning : Temporal Difference learning

Acknowledgements

An optional section, of at most 1 **page**

It is common practice (although totally optional) to acknowledge any third-party advice, contribution or influence you have found useful during your work. Examples include support from friends or family, the input of your Supervisor and/or Advisor, external organisations or persons who have supplied resources of some kind (e.g., funding, advice or time), and so on.

0.0.1 Plan

1. Carl Henrik Ek - Validating my understanding of deep reinforcement learning

Chapter 1

Contextual Background

This chapter explains on a high level what path tracing is and how it accurately simulates light transport. Then importance sampling ray directions in light transport simulation is discussed, and how it can potentially reduce the number of zero contribution light paths and the associated benefits with this. Temporal difference learning as a branch of reinforcement learning is then introduced, along with how it can be used in importance sampling ray directions towards light sources. With a conceptual overview of theory my work is based on, I take a look at recent work which contributes to real-time accurate light transport simulation which my work aims to contribute to. Finally, an overview of the objectives and significant challenges of my investigation are described.

1.1 Path Tracing for Light Transport Simulation

Path Tracing is a Monte Carlo method for rendering photo-realistic images of 3D scenes by accurately approximating global illumination [13]. Figure 1.1 summarises on a high level how forward Path tracing produces a 2D image of a 3D scene. For each pixel multiple rays are shot from the camera through the pixel and into the scene. Any ray which intersects with an area light terminates, otherwise a new direction is sampled for the ray and it is fired again. This process is repeated until all rays have intersected with an area light, at which point the pixel colour value can be found by averaging the colour estimate of each ray fired through that pixel. Each rays colour estimate is calculated based on the material surface properties it intersects with before intersecting with the light and the intersected area lights properties. The more rays shot through each pixel (also known as samples per pixel), the higher the quality of the rendered image becomes, but at a higher computational cost.

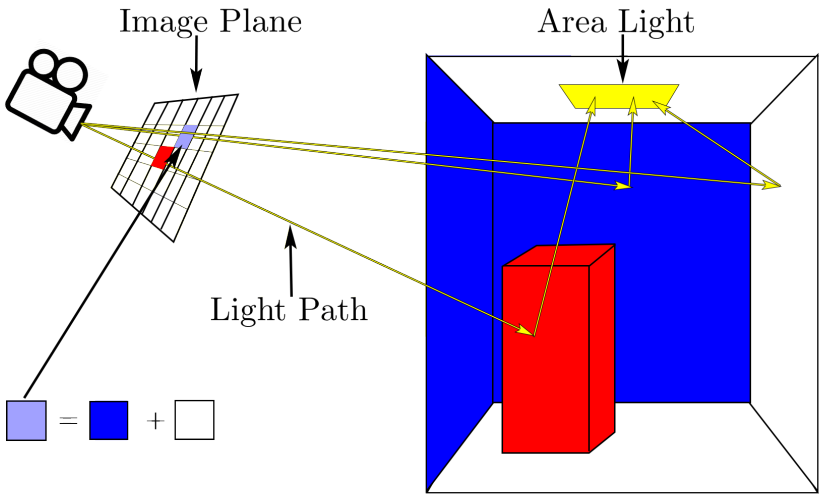
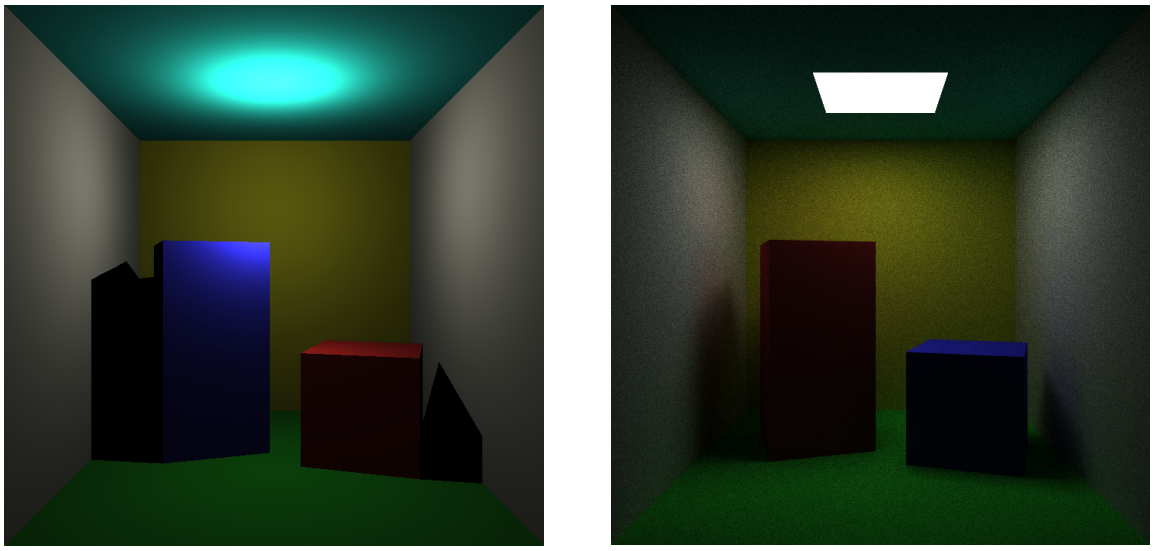


Figure 1.1: An illustration of path tracing, where three light paths are traced from from the camera through a pixel, to the light source in a simple 3D scene.

Path tracing simulates global illumination, meaning it accounts for both direct and indirect illumination. Direct illumination being light paths emitted from a light source, which reflect off exactly one surface before reaching the camera in the scene . Whereas in indirect illumination, light paths reflect 2 or times before reaching the camera. In Figure 1.2, an identical scene is shown with only direct illumination (left) and the other with global illumination (right). The globally illuminated scene displays a range of effects due to Path tracings ability to accurately simulate light transport, which is not the case for the directly illuminated scene. Where light transport simulation refers to firing and summing up the contributions of light transport paths that connect from the camera to light sources [31], such as those displayed in Figure 1.2. For example, effects such as; (a) colour bleeding which is clear on the white walls by the boxes. (b) Soft shadows of the boxes silhouette. (c) Indirect diffuse lighting from light transport simulation causes the shadow of the box to not be pitch black.



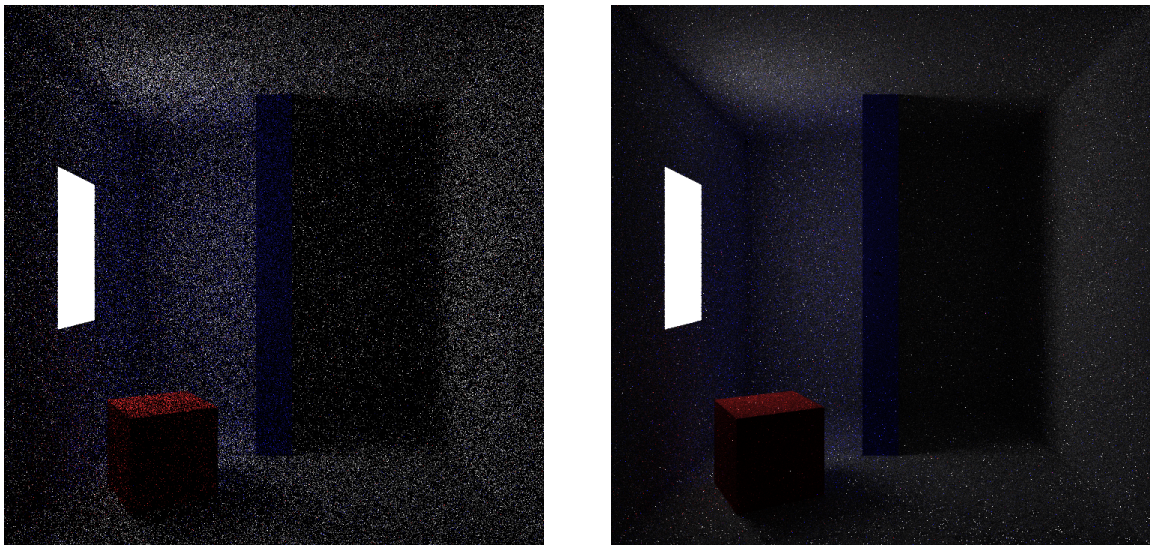
(a) Direct Illumination (b) Global Illumination

Figure 1.2: Two renders of the Cornell Box, where the left is directly illuminated and the right is globally illuminated.

Light transport simulation methods are able to produce many complex light transport effects by a simple single pass of a rendering algorithm. This allows artists to increase productivity and perform less manual image tweaking in the production of photo-realistic images. Due to this, the Computer Graphics industry has seen a large resurgence in research and usage of light transport simulation rendering methods in the past decade [33].

My work in this thesis focuses on developing and assessing importance sampling techniques using Temporal Difference learning methods for light transport simulation in forward Path tracing. In particular, More specifically, for any intersection point in a 3D scene, I attempt to create an AI agent that learns and samples in directions light is coming from, reducing the total number of zero contribution light paths. A zero contribution light path is one whose estimated colour values are almost zero for all (R, G, B) components, hence, they contribute almost no visible difference to the rendered image. We should instead focus our sampling on light paths which do contribute to the image, reducing the noise in pixel values and bringing them closer to their true values for the same number of sampled rays per pixel. Meaning, Importance sampling can reduce the number of rays needed to be sampled per pixel in order to receive a photo-realistic (also known as converged) image from Path tracing. An example of this reduction in noise can be see in 1.3, where the default forward path tracers output is compared to Nvidia's on-line reinforcement learning Path tracer which uses Importance sampling. Note, any light transport simulation algorithm [28, 31] can benefit from the Temporal DIfference learning schemes which will be described, as they are all derived from what is known as the rendering equation. This equation is used as a mathematical basis of modelling light transport.

It is paramount that Importance sampling Path tracing algorithms continue to accurately simulates



(a) Default Forward Path Tracer

(b) Expected Sarsa Path Tracer

Figure 1.3: Two renders of a simple room using 16 sampled light paths per pixel. Where one does not use importance sampling in the construction of light paths (left), and the other does so based on a reinforcement learning rule (right). A clear reduction in image noise can be seen.

global illumination in order to produce photo-realistic images in a single rendering pass, as this is the major selling point of Path tracing over other methods. Therefore, I will also be assessing the accuracy of the global illumination approximation made by the Importance sampling algorithms compared to that of the naive forward Path tracing algorithm.

1.2 Temporal Difference Learning for Importance Sampling Ray Directions

There are three important unanswered questions up to this point; a) what is temporal difference learning? b) How can temporal difference learning methods be used to importance sample new ray directions for a given intersection point in the scene? c) Why use temporal difference learning methods over other Importance sampling methods to do so?

1.2.1 What is Temporal Difference learning?

Temporal difference learning, which I will refer to from here on as TD-learning, are a set of model free Reinforcement learning methods. Firstly, Reinforcement learning is the process of an AI agent learning what is the best action to take in any given state of the system it exists within, in order to maximise a numerical reward signal [56]. The AI agent is not told which actions are best to take in a given state, but instead it must learn which ones are by trialling them and observing the reward signal. Actions taken may not only affect the immediate reward, but all subsequent rewards received for taking future actions. For example, picture a robot rover whose duty it is to explore the surrounding area as much possible. A state in this case is a position in the world it is exploring, and its action are the directions to move in for a given distance. If it discovers a new area, it receives a positive reward signal. Now, if the robot chooses to explore a given area it may not be able to get back from, say a canyon, the robot is limited to searching areas reachable from the canyon. Hence, all subsequent reward signals are limited to what can be received from exploration of the canyon, compared to not entering the canyon and exploring areas which can be returned from first.

1.2.2 Temporal Difference learning methods for Efficient Light Transport Simulation

One of my main aims to reduce the number of zero contribution light paths sampled in Path tracing by the use of TD learning methods. In order to do so I must formulate the problem a reinforcement

learning problem, which is done in detail in Chapter 2. However for a conceptual overview it suffices to explain what a state, action, and reward signal will be in the case of light transport simulation within Path tracing:

- **State**: A 3D intersection position in the scene for a given ray to sample the rays next direction from.
- **Action**: Firing the ray in a given direction (3D vector) from the current state.
- **Reward Signal**: The amount of light incident from the direction the ray was sampled in.

In this reinforcement learning setting, we can use TD-learning methods to create an AI agent which learns by taking different actions in different states and observes their reward signals to find out for each state which actions have the highest valuations. By then converting the action space into a probability distribution weighted by each actions learned valuation, the AI agent will more likely sample non-zero contribution light paths, reducing noise in rendered images. Note, the term valuation means the total expected reward for taking a given action, meaning valuation not only accounts for the immediate reward, but the expected reward for taking all future actions to come until the ray intersects with a light. Also, for the proposed AI agent, current actions can affect future rewards, as when the ray intersects a surface it looses some energy. Therefore, future rewards received after many intersections will be discounted compared to the reward of received immediately to match this behaviour. This means the agent will aim to minimise the average number of intersection a ray makes before intersecting with a light source, making it a good metric to test evaluate against to determine how well the AI agent is performing.

1.2.3 Why use Temporal Difference Learning for Importance Sampling?

Traditional Importance sampling techniques for path tracing do not take into account the visibility of the object from light source. A light blocker is shown in 1.4, where the blocking object stops rays from directly reaching the light. Due to the unknown presence of blockers, traditional importance sampling methods can fail to avoid sampling zero contribution light paths. Therefore, scenes which are significantly affected by blockers will not receive the benefits from traditional Importance sampling and can even benefit more from an uniform sampling scheme [48].

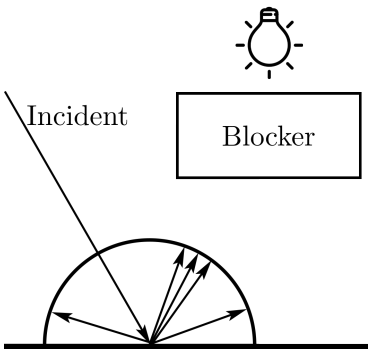


Figure 1.4: An illustration of a light blocker for an importance sampling scheme which does not consider visibility. Each arrow represents a possible direction the light path will be reflected in. Clearly the reflected light path is likely to hit the blocker increasing the likelihood of it becoming a zero-contribution light path.

Temporal difference learning methods are better equipped to tackle this problem [16]. As the AI agent described in the previous section learns which directions light is coming from in the scene and concentrates its sampling towards these directions. Directions leading to blockers will have a low value, hence it is unlikely the AI agent will sample rays in these directions.

1.3 Motivation

Rendering time of my graphics engine is not something I have tried to heavily optimise. I instead focus on producing higher quality images using the same number of samples per pixel in light transport simulation in hope that future work will find ways of optimising my methods for speed. Therefore, my work still aims to contribute to the wider goal seen in computer graphics to use accurate light transport simulation in the rendering of photo-realistic images for complex scenes in real-time. Speeding up the methods I use is a large topic in itself, requiring a deep investigation into the best software, hardware, and parallel programming paradigms to use.

1.3.1 Real time Rendering using Accurate Light Transport Simulation

The motivation for using accurate light transport simulation in real-time comes from the clear superior visual quality of images rendered using this techniques, compared to that of scanline methods which are currently used. Where scanline rendering, also known as rasterizing, is the current computer graphics industry standard method for real-time rendering. Not only are renders for a wide range of scenes clearly superior from methods which accurately simulate light transport, but they also scale far better with the number of polygons used to build the scenes surfaces. Therefore, scanline rendering for scenes with extremely complex geometry in real-time is currently not and option. Accurate light transport simulation methods therefore have great potential to be used in ultra realistic simulations for applications such as scenario planning and virtual reality learning environments [46]. Also, many games sell realism as one of their most important features, therefore developing photo-realistic graphics in real-time has clear economic incentive for the video games industry which was valued at over $136 by the end of 2018 [10]. An economic incentive can also be seen for the film industry, where reductions in render times lead to a direct saving on compute time, as well as the hardware required to render full length films.

1.3.2 Recent Developments

Due to the incentives, a large amount of research and investment has been focused on purpose built hardware and Deep learning post-processing methods in an attempt to bring accurate light transport simulation into real-time. NVIDIA's Turing Ray Tracing Technology [45] represents a significant leap in the hardware to support light transport simulation. It allows for real-time graphics engines to be a hybrid of both scanline rendering, and ray-tracing. The 20 series Turing GPU architecture has significantly improved the speed of ray-casting for light transport simulation, and has the capacity for simulating 10 Giga Rays per second. However, using this hardware alone with current rendering methods is not enough to perform accurate light transport simulation for complex scenes in real-time.

Post-processing methods are designed to take a noisy input image produced by a render which simulates light transport, and then reconstruct the image to remove the noise present in the image. Generally these methods rely on pre-trained deep neural networks to reconstruct the image far quicker then it would take for the renderer to produce an image of the same visual quality [7]. Once again NVIDIA has made significant advancements in this area with NVIDIA OptiX AI Accelerated Denoiser, which is based on their newly designed recurrent denoising autoencoder [11]. OptiX has been successfully integrated in to many of the top rendering engines which accurately simulate light transport, such as RenderMan [12] and Arnold [22]. Whilst post-processing has significantly reduced the number of samples required to render photo-realistic images, there is still more work to be done to produce these images in real-time.

By using importance sampling by TD learning to reduce the number of samples required for accurate light transport simulation, the same standard of noisy image can be fed into an AI accelerated denoiser with fewer samples per pixel in light transport simulation. Running a rendering engine optimised in this way on purpose built hardware could make accurate light transport simulation for rendering photo-realistic images closer than it ever has been to real-time.

1.4 Challenges and Objectives

As previously mentioned, there already exists an example of TD learning used for importance sampling ray directions in a forward Path tracer [16]. However, further methods of analysis need to be conducted

upon this new method to determine its performance for reducing the number of zero contribution l ight paths for different scenes with different settings. It is difficult to assess this as there are infinitely many scenes the method can be used to render, so coming to a clear conclusion is difficult. Another difficult task is that of designing an algorithm for an AI agent to learn what are the favourable directions to sample in a scene are using the deep Q-learning method. This includes some important unanswered questions, such as; is it possible for a deep neural network to model all Q values for a continuous scene space? If so, what is a suitable network architecture? All of which I will describe in more depth in Chapter **??**. Then the actual task of implementing such an algorithm in a graphics engine written from scratch is non-trivial due to the technologies which will need to be combined together. The algorithm must also run fast enough to collect large amounts of data from, otherwise a justified conclusion on its performance cannot be made. Therefore, the algorithm will have to be parallelized and run on a GPU.

As previously mentioned, my main goal is to investigate the ability of two different temporal difference learning algorithms ability to reduce the number of zero contribution light paths in path tracing, whilst still accurately approximating global illumination. Which can be broken down in to the following objectives:

1. Reimplement Nvidia's state of the art on-line Temporal Difference learning Path Tracer in order to further investigate its ability to reduce the number of zero contribution light paths.

2. Design and implement an on-line Deep Q-Learning variant of the Path tracing algorithm and investigate its ability to reduce the number of zero contribution light paths sampled.

3. Assess both Nvidia's state of the art on-line Temporal Difference learning Path tracer, and the Deep Q-Learning Path tracer' on their ability to accurately simulate Global Illumination.

Chapter 2

Technical Background

The goal of this section is to give you as the reader a deep understanding of the technical concepts which build on top of one another as a way to reduce image noise in path tracing. Initially I introduce Monte Carlo integration to approximate and integral, as well as importance sampling and how it reduces variance in the approximation. Monte Carlo integration is the fundamental method which path tracing relies on, I describe this in detail within the section on Monte Carlo path tracing and physical laws path tracing relies on. With the method of path tracing for rendering images clear, I introduce reinforcement learning, as this is what I will be using for the mathematical basis of the importance sampling methods which I use. Here, the crucial concept of a learning the optimal value function is introduced and methods on how to do so. Finally, a TD-learning rule is applied to the context of light transport simulation, and by doing it opens the potential to build more efficient importance sampling methods for light path construction.

2.1 Monte Carlo Integration and Importance Sampling

The theory of Monte Carlo integration and importance sampling underpins how the noise in images rendered by path tracing can be reduced when using the same number of sampled rays per pixel. Therefore, it is necessary to have a good understanding of Monte Carlo integration and its properties, as well as importance sampling before applying it to path tracing.

2.1.1 Monte Carlo Integration

Monte Carlo Integration is a technique to estimate the value of an integral, Equation 2.1 represents this integral for a one-dimensional function f.

$$F = \int_a^b f(x)dx \tag{2.1}$$

The idea behind Monte Carlo integration is to approximate the integral by uniformly sampling points (x_i) to evaluate the integral at, and then averaging the solution to the integral for all the sampled points. More formally, basic Monte Carlo integration approximates a solution to this integral using the numerical solution in Equation 2.2. Where $\langle F^N \rangle$ is the approximation of F using N samples.

$$\langle F^N \rangle = (b-a)\frac{1}{N}\sum_{i=0}^{N-1} f(x_i) \tag{2.2}$$

$$\langle F^N \rangle = \frac{1}{N}\sum_{i=0}^{N-1} \frac{f(x_i)}{\frac{1}{(b-a)}} = \frac{1}{N}\sum_{i=0}^{N-1} \frac{f(x_i)}{pdf(x_i)} \tag{2.3}$$

An important property of Monte Carlo Integration is that it produces an unbiased estimate of an integral, meaning average of $\langle F^N \rangle$ is exactly the true value of the integral,F for any N [37]. This is presented in Equation 2.4, where p_i is the probability of a given of a given approximation $\langle F^N \rangle$. Basic Monte Carlo integration only produces a non-bias estimate when sample points x_i are randomly sampled from a uniform distribution. To extend this to Generalized Monte Carlo integration where sample points may be sampled from any distribution, the function evaluated at point x_i must be divided by the probability density function (pdf) evaluated at x_i. This is known as generalized Monte Carlo integration

and is shown in Equation 2.3, which from here onwards I will refer to as Monte Carlo integration. Dividing by the *pdf* ensures the estimate $\langle F^N \rangle$ is unbiased, as areas with a high *pdf* will be sampled far more, but their contribution weighting ($\frac{1}{pdf}$) to final estimate will be lower. Whereas areas with a low *pdf* will be sampled less, but their contribution weighting to the final estimate will be higher to offset this.

$$\mathbf{E}[\langle F^N \rangle] = \sum_{i=0}^{k-1} \langle F^N \rangle_i * p_i = F \tag{2.4}$$

Another important property of Monte Carlo integration is that by the law of large numbers, as the number of samples (N) approaches infinity, the probability of the Monte Carlo approximation ($\langle F^N \rangle$) being equal to the true value of the integral (F) converges to 1. This law is stated in Equation 2.5. By this property Monte Carlo Integration works well for multidimensional functions, as convergence rate of the approximation is independent of the number of dimensions, it is just based on the number of samples using in the approximation. Whereas this is not the case for deterministic approximation methods, meaning they suffer from what is known as the curse of dimensionality. For path tracing, the integral which is approximated is a 2 dimensional function, hence Monte Carlo integration is used.

$$Pr(\lim_{N \to \infty} \langle F^N \rangle = F) = 1 \tag{2.5}$$

The standard error of the Monte Carlo integration approximation decreases according to Equation 2.7. Where the standard error describes the statistical accuracy of the Monte Carlo approximation. Where σ_N^2 is the variance of the solutions for the samples taken, and is calculated by Equation 2.6 using the mean of the solutions for the samples taken (μ). Due to Equation 2.7, in practice four times as many samples are required to reduce the error of the Monte Carlo integration approximation by a half. Also, the square root of the variance is equal to the error of the approximation, so from here on when I refer to reducing the variance I am also implying a reduction in the error of the approximation.

$$\sigma_N^2 = Var(f) = \frac{1}{N-1} \sum_{i=0}^{N} (f(x_i) - \mu)^2 \tag{2.6}$$

$$\text{Standard Error} = \sqrt{Var(\langle F^N \rangle)} = \sqrt{\frac{\sigma_N^2}{N}} = \frac{\sigma_N}{\sqrt{N}} \tag{2.7}$$

2.1.2 Importance Sampling for Reducing Approximation Variance

Currently I have only discussed Monte Carlo integration by sampling points x_i to solve the integral using a uniform distribution. However the purpose of introducing Equation 2.3 was to create a custom *pdf* which can be used for importance sampling to reduce the variance of the Monte Carlo integration approximation. To understand how and why importance sampling works, first observe Figure 2.1 where a constant function is given with a single sample point evaluated for $f(x)$. This single sample is enough to find the true value for the area beneath the curve i.e. integrate the function with respect to x. This is shown in Equation 2.8, where $p = f(x)\ \forall x \in \mathbb{R}$.

$$\langle F^N \rangle = (b-a)\frac{1}{N} \sum_{i=0}^{N-1} f(x_i) = (b-a)\frac{1}{N} \sum_{i=0}^{N-1} p = pb - pa \tag{2.8}$$

However, Figure 2.2 requires many samples to accurately approximate the integral when sampling from a uniform distribution. This is due to the functions complex shape, meaning many samples are required to calculate the are beneath the curve within the Monte Carlo integral approximation. Generally, it requires fewer samples to approximate a function which is closer to being constant function [37].

Most functions are not constant, however it is possible to turn any function into one, and this is exactly what can be done within Monte Carlo integration. To convert a function f to a constant function, a function f' can be introduced which produces the same output as f for every input, but scaled by a constant c [52]. The function f is then divided by f' to produce a constant function, as shown in Equation 2.9.

$$\frac{f(x)}{f'(x)} = \frac{1}{c} \tag{2.9}$$

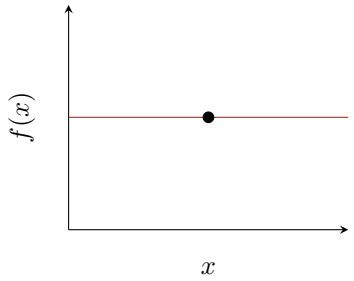


Figure 2.1: Constant Function with a sample point

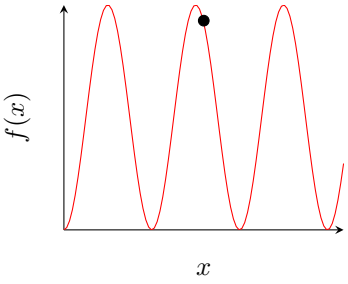
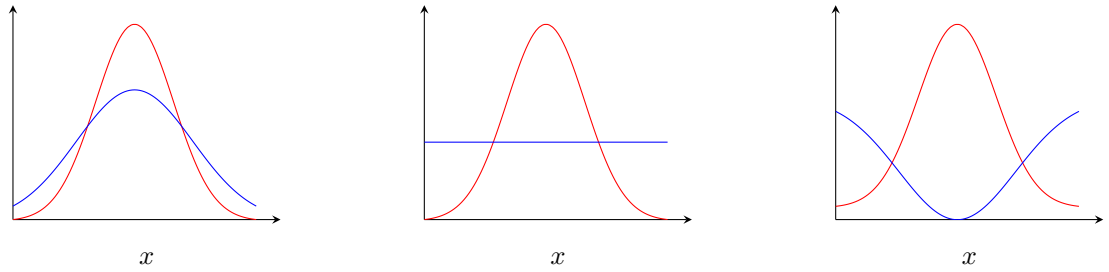


Figure 2.2: Non-linear Function with a sample point

This can be applied to Monte Carlo integration stated in Equation 2.3, by choosing a probability density function (*pdf*) which produces the same output as f for all inputs, but divided by some normlazing constant factor c, keeping *pdf* as a probability distribution. Therefore, we are able to calculate the true value of the integral through Monte Carlo integration as shown in Equation 2.10. Where it turns out $\frac{1}{c}$ is true value for the integral in Equation 2.1.

$$\langle F^N \rangle = \frac{1}{N} \sum_{i=0}^{N-1} \frac{f(x)}{pdf(x)} = \frac{1}{N} \sum_{i=0}^{N-1} \frac{f(x)}{cf(x)} = \frac{1}{N} \sum_{i=0}^{N-1} \frac{1}{c} = \frac{1}{c} \tag{2.10}$$

For most cases it is not possible to know the correct probability density function which can convert the Monte Carlo integration problem into integrating a constant function. However, if one has prior knowledge regarding 'important' regions of the functions input space, it is possible to create a probability density function whose shape matches f more closely then a uniform probability distribution. By Important areas of the function input space, I mean areas of the input space which produce a large contribution to the integral of the function. For example in Figure 2.3a, the most important regions are around the top of the functions peak.



(a) Importance sampling reducing variance

(b) Uniform sampling

(c) Importance sampling increasing variance

Figure 2.3: Graphical representation of a function $f(x)$ (red) and the corresponding probability density function $pdf(x)$ (blue) used in the Monte Carlo integration approximation for the integral of $f(x)$.

As previously explained, Figure 2.3a represents a probability density function which has a similar shape to the function which is being integrated. Therefore the variance in the Monte Carlo integration approximation will be lower then that of the uniform distribution shown in Figure 2.3b. Figure 2.3c presents an example where the created probability density function does not resemble the shape of the function which is being integrated. Using this *pdf* in Monte Carlo integration would significantly increase the variance in the approximation compared to that from a uniform *pdf* shown in Figure 2.3b. This is due to regions which have high importance according to the *pdf* contribute a low amount to the integral of the function f, causing the variance in the Monte Carlo integration approximation to rise.

2.2 Monte Carlo Path Tracing

In 1986 James Kajiya introduced the rendering equation and with a it a Monte Carlo integral approximation to the equation [29]. This Monte Carlo approximation is essentially what is known as today as

Monte Carlo Path Tracing. Here, I will give a detailed explanation of the rendering equation and how Monte Carlo Path Tracing approximates the equation by accurately simulating light transport. As Path tracing is a involves a Monte Carlo integral approximation, importance sampling can be used to reduce the variance in its approximation as described in Section 2.1.2.

2.2.1 The Rendering Equation

Equation 2.11 is the rendering equation. It calculates the radiance incident from a point x at a given viewing angle ω. Radiance indicates the power of light emitted, transmitted, reflected or received by a surface from a given viewing angle, with units watts per steradian per square metre ($W \cdot sr^{-1} \cdot m^{-2}$). Therefore, by placing a camera in a scene, the radiance incident on the lens from a given surface determines the cameras perceived colour and power of light incident from the surface. These values are used to calculate pixel values in computer image generation. The equation states how to correctly perform light transport simulation for rendering, and in turn how to accurately simulate global illumination. Therefore, methods which can accurately approximate the rendering equation for any given scene can convert the incident radiance into pixel values to produce realistic rendered images of any given scene. The exact details of how this is done will be described in the next section on the forward path tracing algorithm.

$$\underbrace{L_o(x,\omega)}_{\text{Outgoing}} = \underbrace{L_e(x,\omega)}_{\text{Emitted}} + \underbrace{\int_\Omega L_i(h(x,\omega_i),-\omega_i) \cdot f_r(\omega_i,x,\omega) \cdot \cos(\theta_i) d\omega_i}_{\text{Reflected}} \tag{2.11}$$

Where:

$L_o(x,\omega)$	= The total outgoing radiance from a 3D point x, in the direction ω
$L_e(x,\omega)$	= The emitted radiance from the point x
Ω	= Hemisphere centred around the normal n of the surface, containing all possible angles ω_i
$L_i(y,-\omega_i)$	= The radiance incident from the intersected position y in direction ω_i
$h(x,\omega_i)$	= Returns the closest intersected position by firing a ray from x in direction ω_i
$f_r(\omega_i,x,\omega)$	= The BRDF, describing the proportion of light reflected from ω_i in direction ω
$cos(\theta_i)$	= Cosine of the angle between surface normal at point x and the direction ω_i

The rendering equation is based on the physical law of the conservation of energy, where the outgoing radiance in a given direction (L_o) from a point is equal to the emitted light (L_e) from the point in the direction, plus the reflected light (the integral) from that point in the direction. The emittance term L_e is simple, it is the light emitted the point x which has been intersected, if this is non-zero a light source has been intersected with. However, the reflected light which is represented by the integral is generally analytically intractable, as it involves summing the contribution of incoming radiance from infinitely many directions in the hemisphere Ω around the point x (L_i). Also, the term L_i is recursive [19], as to calculate the radiance incident in the direction ω_i from some hit-point say $y = h(x,\omega_i)$, a solution is required for $L_o(y,\omega)$. This concept is represented Figure 2.4.

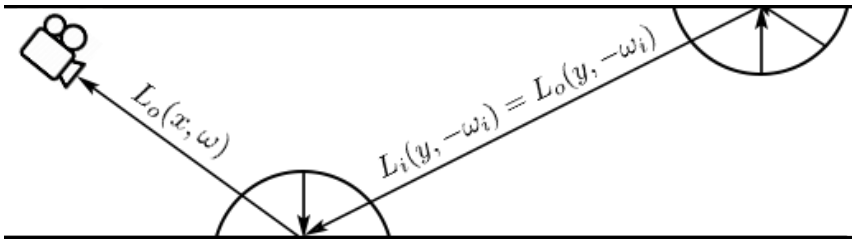
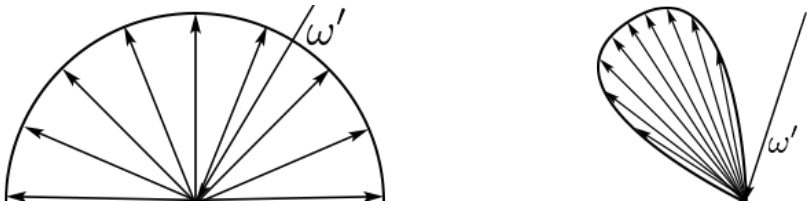


Figure 2.4: A diagrammatic representation of the recursive nature of the rendering equation. The outgoing radiance (L_o) in a given direction ω from a point x requires an estimation of the incident radiance coming from all angles in the hemisphere around the point, that is $L_i(h(x,\omega_i),-\omega_i) = L_i(y_i,-\omega_i)\ \forall\omega_i \in \Omega$. To calculate $L_i(y_i,-\omega_i)$ is identical to calculating the outgoing radiance $L_o(y_i,-\omega_i)$ as we assume no radiance is lost along a ray line, hence the L_o is a recursive function.

The f_r term in Equation 2.11 is known as the bidirectional reflectance distribution function (BRDF). On a high level, the BRDF describes how a surface interacts with light [23]. Every surface has a BRDF which determines when a ray intersects with that surface at a given incident direction ω, the ratio of

reflected radiance in direction ω. Therefore, querying the BRDF for a surface at point x with incident ray direction ω' and given reflected direction ω, that is $f_r(\omega', x, \omega)$, a single scalar value is returned. A diffuse and specular surfaces BRDF's are depicted in 2.5. A diffuse material reflects light almost equally in all directions for any angle of incidence, an example is paper. Whilst for specular materials, incident rays are reflected in a narrow area around the perfect reflection direction, many metals exhibit specular reflections.



(a) Diffuse BRDF (b) Specular BRDF

Figure 2.5: A representation of both a diffuse surface and specular surface BRDF for a given angle of incidence ω'. The surface point is located where all end of the arrows converge. The arrows indicate a subset of direction possible for the incident ray to be reflected in. All possible directions reflected directions for a ray are defined between the surface point and the line , for an incident direction ω'. The further away a point is on the line, the more likely a ray is to reflected in a direction from the surface point to that point on the line. The diffuse surface is equally likely to reflect a ray in any direction. Whereas, the specular surface favour a small subset are of direction in the hemisphere surrounding the surface point.

Another way to think about diffuse and specular materials is do they change in appearance depending on the viewing angle? For example, surface of paper appears to be identical no matter the viewing angle, however a shiny metal ball would appear to reflect what was in front of it which changes depending on the viewing angle, just like a mirror. These differences can be scene in Figure 2.6.

(a) La Table aux Amours, Marble Sculpture

(b) Cloud Gate, Stainless Steel Sculpture

Figure 2.6: Two sculptures, one made from a diffuse material (left) and the other from a specular material.

A scene comprising of only diffuse materials is generally more computationally expensive to simulate, as one has to simulate rays reflecting in all directions around intersections points with surfaces, compared to a specular scene where only a small subset of directions need to be sampled for each intersection. So, from here on whenever a surface or BRDF is mentioned, assume it is diffuse as my descriptions can be extended to specular materials by restricting ray reflections to a limited set of angles.

Finally, as mentioned $cos(\theta_i)$ is the cosine of the angle ω_i between the normal of the point of the surface intersected with and the angle of incidence. The normal of a surface is the normalized vector that is perpendicular to the surface [64]. The $cos(\theta_i)$ is a weighting for reflected radiance from a point, where the larger the angle from the normal the smaller the reflected radiance. This simulates how light is spread across the surface, causing less light to be reflected in a direction which is further away from being perpendicular to the surface. The combined BRDF and cosine term in the rendering equation uphold the physical law of conservation of energy, meaning more radiance cannot be reflected from the surface then incident on the surface. This is formally described by Equation 2.12 [23], where ω_r represents a direction of reflection.

$$\forall \omega_i, \int_{\Omega} f_r(\omega_i, x, \omega_r) \cos(\theta_r) d\omega_r \leq 1 \tag{2.12}$$

2.2.2 Path Tracing

Monte Carlo Path Tracing

In section 1.1 I already gave a high level overview of the how the path tracing algorithm where many light paths are sampled which consist of shooting a ray from the camera, through a pixel and into the scene to calculate a colour estimate. A pixels colour is then determined by averaging all light paths colour estimates. However I did not detail how to get the colour estimate of a light path. This is exactly what the solution to the rendering equation gives, as $L_o(x, \omega)$ gives the outgoing radiance for each sampled light paths initial direction ω and intersection point x. The radiance is then converted into a pixel colour value. Put another way, $L_o(x, \omega)$ is a pixels colour value where ω is the direction of the ray when shot from the camera, through the pixel and into the scene. Then x is the position in the scene the ray first intersects with.

But how does one solve the rendering equation, as often it cannot be done analytically? This is what Monte Carlo integrating in path tracing is used for. Path tracing solves a slightly different form of the rendering equation to that in 2.11. To calculate the reflected radiance at point x in the scene for the angle of incidence ω, it is possible to instead calcualte the integral of all light paths which start at the intersection x and reflect round the scene until a light source is intersected with. The proof behind this is detailed in [25], but conceptually it is simple. Previously the reflected radiance for (x, ω) was given by the integral of the incident radiance on x with respect to the angle of incidence. To calculate this integral one can trace infinitely rays from the intersection point x in all possible directions Ω until they high with a light source, the sum of which gives the total amount of incident light on point x. Therefore, path tracing solves a variant of the rendering equation to estimate $L_o(x, \omega)$ by integrating over all possible light paths starting from x with respect to the surfaces intersected with. It is this integral which is solved via Monte Carlo integration, the details of which are given in Equation 2.13.

$$L_o^N(x, \omega) = \frac{1}{N} \sum_{k=0}^{N-1} L_e(x_0, \omega_0) + (L_i(x_1, -\omega_1) \cdot f_s(\omega_1, x_1, \omega_0) \cdot cos(\theta_{\omega_1}))/\rho_1$$

Such that

$$L_i(x_i, -\omega_i) = \begin{cases} L_e(x_i, \omega_i) + (L_i(x_{i+1}, -\omega_{i+1}) \cdot f_s(\omega_{i+1}, x_{i+1}, \omega_i) \cdot cos(\theta_{\omega_{i+1}}))/\rho_i & \text{if } x_i = \text{Light Source} \\ L_e(x_i, \omega_i) & \text{otherwise} \end{cases} \tag{2.13}$$

Where:

x_i = Intersection location of the light path after i reflections in the scene
ω_i = Direction of the light path after i reflections in the scene
ρ_i = Probability density function over reflected ray directions for position x_i and ω_i angle of incidence

In Equation 2.13 recursive L_i is still present, but the recursion is terminated when the light path intersects with a light source. By the law of large numbers in Equation 2.5, the larger the number of sampled light paths (N), the closer each pixels approximation will be to the pixels true value as a result of solving the rendering equation. As known from section 2.2.1, the rendering equation follows the physical law of energy conservation, and due to this it accurately models light transport simulation for global illumination. Therefore, the more samples used in the Monte Carlo approximation in Equation 2.13, the lower the amount of noise in the image [13]. An example of this concept applied to a simple forward path tracer is shown in Figure 2.7, and the pseudo code for this forward path tracer to produce a single rendered image is given in Algorithm 1.

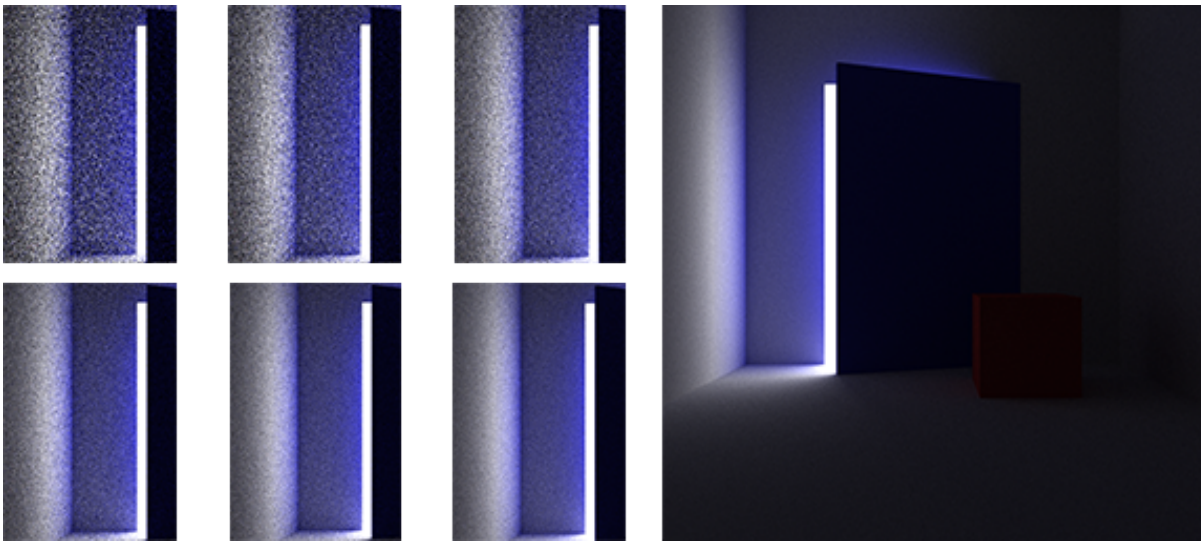


Figure 2.7: An indirectly illuminated scene from a default path tracer. The grid of image sections represent an increasing number of samples per pixel (SPP), beginning in the top left with 16 SPP, to the bottom right with 512 SPP. The full image on the right is a reference image with 4096 SPP where the Monte Carlo approximation has almost converged for pixel values.

Algorithm 1: Forward Path Tracer. Given a camera position, scene geometry, this algorithm will render a single image by finding the colour estimate for each pixel using Monte Carlo path tracing. Where N is the pre-specified number of sampled light paths per pixel.

```
Function renderImage(camera, scene)
    for i = 1 to N do
        for p in camera.screen do
            ray ← initializeRay(p, camera)
            for j = 1 to ∞ do
                (y, n, L_e) ← closestIntersection(ray, scene)
                if noIntersection(y) or areaLightIntersection(y) then
                    ray.throughput ← ray.throughput · L_e
                    updatePixelColourEstimate(p, ray.throughput)
                    break
                end
                (ω_i, ρ_i, f_s) ← sampleRayDirRandomly(y)
                ray.throughput ← ray.throughtput · f_s · (ω_i · n)/ρ_i
                ray ← (y, ω_i)
            end
        end
    end
end
```

Importance Sampling in Path Tracing

As path tracing is a Monte Carlo method for solving the rendering equation, Importance sampling can be applied in order to reduce the variance pixel colour estimates. In section 2.1.2 it was shown that by using a probability density function (*pdf*) which closely matches the shape of the function being integrated, the variance in the Monte Carlo estimate is significantly reduced. Applying this to Equation 2.13, the term ρ_i which represents the probability density function for sampling the next ray direction at intersection location x_i with angle of incidence ω_i. Currently you can assume that the probability density function ρ_i is uniform. But this can be modified with prior knowledge regarding which directions are more important for continuing a light path in, where an important direction is one which leads to a high contribution of radiance to the pixel estimate.

The question now is, can one have any knowledge for which directions contribute the most radiance to the pixels colours value? The answer is yes, and there has been a large amount of research in this which resides in the topic of light transport simulation. The simplest example lies within the rendering equation itself, $cos(\theta_i)$. As previously discussed, this term acts as a weighting for the radiance contribution of outgoing light paths. So, the probability density function ρ_i can also be weighted by $cos(\theta_i)$, which is likely to reduce the pixel value variance. There exists many other methods of retrieving knowledge from the scene to use in importance sampling during rendering. For example, irradiance cahcing [8], table-driven adaptive importance sampling [14], and sequential Monte Carlo adaptation [47]. However as discussed in section 1.3, these previous methods do not effectively reduce the number of zero contribution light paths, meaning their ability to image noise for certain scenes is very limited. Instead, Nvidiai proposed that reinforcement learning can be used for this [16] which is the main inspiration for my work. In the proceeding sections I will discuss in detail how it is possible to apply reinforcement learning for importance sampling in light path construction.

Existing Methods for Importance Sampling

2.3 Reinforcement Learning and TD-Learning

Now that it is clear how Importance sampling light paths can be used to reduce variance in Monte Carlo path tracing, it is time to introduce the concept of reinforcement learning as I will be using this to gain knowledge for this Importance sampling. This section aims to give a quick introduction to reinforcement learning and TD-learning to cover all of the background material of the learning methods I will be using, before describing how they are applied to path tracing in the next section.

2.3.1 Markov Decision Processes

Reinforcement learning is one of the three archetypes of machine learning and it is concerned with finding what action should be taken in a given situation, in order to maximise a numerical reward [56]. This problem is formalized by a finite Markov Decision Process (MDP), which is designed to capture the most important aspects of the problem a learning agent faces when interacting over time with its environment to achieve a goal. A MDP is summarised in 2.8 and can be described in terms of the following:

- **Agent** - The learner and decision maker which takes an action A_t in an observed state S_t (where t is the current time step), receiving an immediate numerical reward R_{t_+1} and the next observed state S_{t+1}
- **Environment** - What the agent interacts with when taking an action A_t in state S_t and produces both R_{t+1} & S_{t+1}

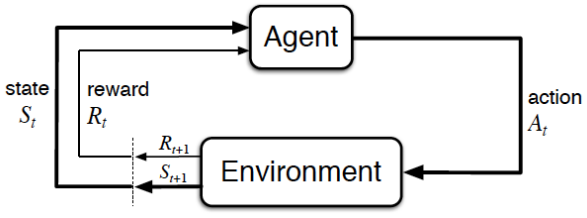


Figure 2.8: Markov Decision Process [56]

An MDP comprises of the following the following tuple:

$$(\mathcal{S}, \mathcal{A}, p, \gamma)$$

Where:

$\mathcal{S}$ = The set of all states
$\mathcal{A}$ = The set of all actions
p = Probability of receiving reward r and state s' when in the previous state s and action a was taken
γ = The discount factor which makes the agent value immediate rewards higher than later ones

An important detail of an MDP which makes it far easier to implement in practice is that any problem modelled by an MDP assumes the Markov property.

> "The future is independent of the past, given the present." - *Hado van Hasselt, Senior Research Scientist at DeepMind* [60]

This is expressed mathematically for an MDP in equation 2.14. Put simply, the Markov property means the current state captures all relevant information from the history of all previous states the agent has experienced, meaning the history is not needed.

$$p(R_{t+1} = r, S_{t+1} = s'|S_t = s) = p(R_{t+1} = r, S_{t+1} = s'|S_1, .., S_{t-1}, S_t) \tag{2.14}$$

2.3.2 Goals and Rewards

The goal thought of for a reinforcement learning agent can change significantly depending on the problem, for example in the case of a game it may be to maximise the total score in one play-through. Or for a robotic space rover it may be to discover the most amount of unseen terrain. However, in terms of an MDP all AI agents goals a described as maximising the total amount of cumulative reward received. This is more formally described by the reward hypothesis [56]

> Any goal can be formalized as the maximisation of the expected value of the cumulative sum of a received scalar reward signal.

Once again in the case of an agent learning the best possible action to take for any state in the game (known as the optimal policy), a reward signal could be the points gained by making a certain move. Therefore, to maximise the expected return would be to maximise the number of points received in a play-through. The return is formally defined in Equation 2.15 in terms of a reward sequence combined with the discount factor, which as previously mentioned trades off later rewards for more immediate ones. If a discount factor (γ) is closer to 1 the agent is said to be far sighted, as it gives future rewards a high weighting. Whereas a myopic agent is one which has a discount factor closer closer to 0, as it gives a lower weighting to future rewards for their contribution towards the return G_t [60].

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.15}$$

This formulation works well if the agents interactions with the environment break down easily into sub-sequences [56], where an agent starts in one of a given set of starting states and takes a series of actions to reach a terminal state. From the terminal state the agent can be reset to one of the starting states to begin learning once again. This applies to path tracing, where the terminal state is one in which the light path has intersected with a light, but this will be discussed in detail in Section 2.4.

2.3.3 Value Function and Optimality

All reinforcement learning algorithms I will be considering involve the concept of a value function. There are two kinds of value functions, one which determines the value of being in a given state, the other determines the value of being in a certain state and taking a certain action, known as a state-action pair. The methods I consider are those which use state-action pair value functions, where the value of a state-action pair is defined in terms of the expected return from that state-action pair.

An agent follows a policy π, which determines how the agent will act in a given state. Formally, a policy is a mapping from states to probabilities of selecting a particular action. When an agent is following policy π at time t, then $\pi(a|s)$ is the probability that $A_t = a$ if $S_t = s$. Reinforcement learning algorithms state how an agents policy changes from experience.

The value of a state-action pair (s, a) under a policy π, is given in Equation 2.16 denoted as $q_\pi(s, a)$,. This value function is commonly known as the action value function for policy π. Stating 'under policy π' is important as the value of a given state-action pair depends upon the actions we take onwards from taking action a in state s due to π. $\mathbf{E}_\pi$ denotes the expected value of a random variable, given that the agent follows policy π. From this, if one were to keep track of the actual returns received for taking a state-action pair, then as the number of times the state-action pair is chosen tends to infinity, the average of the returns will converge on the true expected value of the return for the state-action pair $q_\pi(s, a)$.

$$q_\pi(s,a) = \mathbf{E}_\pi[G_t|S_t = s, A_t = a] = \mathbf{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a] \tag{2.16}$$

Now, if you had an AI agent the best way it could perform would be to maximise the expected reward it receives in an episode. In terms of a policies, this is know as the optimal policy which is said to be better then all other policies and agent can follow. Formally, the optimal policy is π if $\pi \geq \pi'$ for al possible policies π'. Where, $\pi \geq \pi'$ if and only if $q_\pi(s,a) \geq q_{\pi'}(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$. The optimal policy is denoted as π_* and the value function following the optimal policy, which is the optimal value function, is denoted $q_*(s,a)$. The optimal value function is defined in Equation 2.17.

$$q_*(s,a) = \max_\pi q_\pi(s,a) \tag{2.17}$$

$$= \mathbf{E}[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a')|S_t = s, A_t = a] \tag{2.18}$$

Equation 2.18 defines the Bellman optimality equation, which states that the value of a state-action pair under an optimal policy must be equal to the expected return of the immediate reward plus the highest valued state-action pair available from the next state. Intuitively, if the optimal policy is available which is essentially a cheat sheet of what action is most valuable to take in each state. Then the value of a given state-action pair should be equal the immediate reward received by taking the action in the current state, plus the value of the best action to take in the next state given by the cheat sheet/optimal policy. Therefore, if one has the optimal value function, the optimal policy can easily be found by maximising the choice of action a for $q_*(s,a)$ in state s [56].

To summarize, the aim from here on is to build an agent which is able to learn the optimal value function, but whilst this is provably possible, it rarely happens in practice. However, the learning methods I will discuss in the next section on TD-learning are able to find a good value function for light path direction sampling.

2.3.4 Temporal Difference Learning

TD-Learning is combination of Monte Carlo and Dynamic Programming reinforcement learning methods for learning the optimal value function from equation 2.17. I will not discuss the details of Monte Carlo and Dynamic Programming methods as they are not investigated as part of my work. However, the reasoning for choosing to study TD-learning approaches over these two alternative approaches are as follows; TD-learning can perform learning updates of the value function throughout an episode, unlike Monte Carlo approaches which wait until the end [62]. This means TD-learning algorithms can be written in an online learning algorithm [56]. TD-learning can learn directly from experience, as it does not require a true model of the environment in order to learn, unlike Dynamic Programming methods [61]. This means TD-learning is model-free, avoiding the expense of building the true model of the environment. I will now introduce

I will now introduce three different temporal difference learning methods which are required knowledge for the rest of my work.

Sarsa

Sarsa is a on-policy TD method which learns a good state-action pair valuation function $q_\pi(s,a)$. The Sarsa learning rule is presented in Equation 2.19, and I have chosen to present this method first to explain some key concepts TD-learning methods share. Firstly, Q denotes the current value function under policy π, q_π. Therefore the left arrow indicates an update in the value of the current estimate Q. Also notice, how the current estimate is update upon every time step t, this means Sarsa like other TD-learning methods can learn during an episode as previously mentioned. The α term is the current learning rate where $\alpha \in [0,1]$ and γ is the discount factor as previously discussed. Finally, Sarsa performs what is known as bootstrapping in the context of reinforcement learning [56]. Bootstrapping is where the estimate of the valuation function (Q), is updated based on some new data by experience, which is the immediate reward R_{t+1}. As well as, the current estimate of the valuation function Q. Sarsa therefore learns from experience, whereby an action is A_t is taken in state S_t, leading to an immediate reward R_{t+1} which is used to update the current estimate Q.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \tag{2.19}$$

The reasoning behind the name Sarsa is that the method uses every element in the quintuple of events, $Q(S_t, A_t, R_{t+1}, S_{t+1}, A_{t_1})$, which makes up a transition between each time step of state-action pairs. Sarsa is an on-policy TD-learning method, as to choose the next action to take in the next state ($Q(S_{t+1}, A_{t+1})$ the policy is used. Sarsa is proven to converge on the optimal value function q_* when the policy π remains constant.

To make Sarsa learning and TD-learning in general more concrete, imagine a robot with a camera whose goal it is to manoeuvre to the end of a corridor. Each time step is the point when a new frame is rendered on the camera, and the state is the image displayed by the camera. The robots actions consist of moving a short distance in a set of four different direction. If the robot were to learn using Sarsa, the robot would have a large table storing the value of each state-action pair $Q(S_t, A_t)$, which represents the current value function. The robot would then select an action at each time step according to the policy π to receive a reward signal based on its distance to the end of the corridor. The robot would then perform a lookup on the large table indexing with the action it took in the state it was in and perform the update rule in 2.19. This large table representing the current estimate of the optimal value function is also know as a Q-table, where each value in the table is known as a Q-value, $Q(S_t, A_t)$. By following a suitable policy, the robot will over time will keep updating its Q-values to improve its estimate of $q_\pi(s, a)$.

Q-Learning

Q-learning is very similar to Sarsa except it is an off-policy TD-learning algorithm. Also, if all state-action pairs are visited infinitely many times, it is proven that Q-learning can converge on the optimal policy $q_\pi(s, a)$ faster then Sarsa, therefore it is generally a preferred method. The learning rule is displayed in Equation **??**, where rather then following a policy π to select the action to update with, the maximum value of the highest valued action in the next state is selected ($\max_a Q(S_{t+1}, a)$. This means the agent following policy π will still choose its actions in a state based on π, however when it update its valuation function Q, the action chosen to update with may not necessarily be the same as the action chosen. Hence, Q-learning is an off-policy TD-learning algorithm.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{2.20}$$

Expected Sarsa

Expected Sarsa is a TD-learning algorithm which is generally found to be superior to both Q-learning and Sarsa in practice [56]. It is very similar to Q-learning, but instead of using the maximum value of the next state-action pairs it takes the expected value over them. This means Expected Sarsa takes into account how likely each action is under the current policy as shown in Equation 2.22. Where $\pi(a|S_{t+1})$ returns the probability of selecting action a in state S_{t+1}, while following the policy π. Note, Expected Sarsa may be used as either an on-policy of off-policy algorithm, for example if the policy π was set to the greedy-policy used in Q-learning, the learning rule would become identical to that of Q-learning. Therefore, Expected Sarsa generalizes over Q-learning. Expected Sarsa also reduces the variance in the value function approximation compared to that of Sarsa, as the expectation taken over the current valuation estimate for state-action pairs used.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \mathbf{E}[Q(S_{t+1}, A_{t+1})|S_{t+1}] - Q(S_t, A_t)] \tag{2.21}$$

$$\leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{2.22}$$

2.3.5 Exploration vs Exploitation

Up to now I have formally introduced what reinforcement learning is, including what the optimal value function is and different ways to learn it. As for the TD-learning methods presented so far, I have not discussed any details about the kind of policy an agent should use to select actions in the process of learning. Deciding on this policy is very influential on our agents performance, as the agent needs to gather enough information about its environment to make the best overall decisions. Therefore, online

decision-making requires a fundamental choice to be made by the agent every time it chooses to take an action [58]. This is between the following:

- **Exploration**: Maximise the agents performance based on the current knowledge available
- **Exploration**: Gain more knowledge about the environment

This means a good learning strategy for an agent may be to sacrifice short-term performance to maximise it in the long-term. This applies directly to the policy used in TD-learning methods. Initially exploration is the most important to quickly gain a broad amount of knowledge about the environment and opening up more specific areas for further exploration. Then over time the policy should favour exploitation more and more by taking known higher valued actions. An example of this kind of policy is the decaying ϵ-greedy policy [56]. This policy maintains the current value of $\epsilon \in [0, 1]$ and involves sampling a random number $x \in [0, 1]$, then if $x > \epsilon$ exploitation occurs, else exploration. By exploitation it is common practice to choose the current highest valued action, whereas exploration involves choosing one at random. Overtime ϵ is decreased to match the behaviour of increasing exploitation as more knowledge is gained.

2.4 Linking TD-Learning and Light Transport Simulation

I will now to link together the concepts introduced so far in this chapter to derive a way of finding the outgoing radiance from position x in direction ω ($Q(x, \omega)$), based on the TD-learning method Expected Sarsa. I incorperate this learning rule later into Nvidia's online reinforcement learning path tracing algorithm, which uses the learned $Q(x, \omega)$ values for importance sampling directions in light path construction [16].

First, the Expected Sarsa learning rule's summation over the set of all actions $\mathcal{A}$ can be represented as an integral with respect to an action a, as shown in Equation 2.25. This implies the action space is continuous:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1})Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{2.23}$$

$$= (1 - \alpha) \cdot Q(S_t, A_t) + \alpha \cdot \left(R_{t+1} + \gamma \sum_a \pi(a|S_{t+1})Q(S_{t+1}, a) \right) \tag{2.24}$$

$$= (1 - \alpha) \cdot Q(S_t, A_t) + \alpha \cdot \left(R_{t+1} + \gamma \int_{\mathcal{A}} \pi(a|S_{t+1})Q(S_{t+1}, a)da \right) \tag{2.25}$$

Recall the rendering equation from section 2.2.1 describes the radiance in an outgoing direction ω from point x is equivalent to the emitted radiance in the direction plus the reflected radiance in the direction.

$$L_o(x, \omega) = L_e(x, \omega) + \int_\Omega L_i(h(x, \omega_i), -\omega_i) \cdot f_r(\omega_i, x, \omega) \cdot \cos(\theta_i) d\omega_i$$

Now, by matching terms from the rendering equation to the Expected Sarsa learning rule in Equation 2.25, Equation 2.26 is formed. Where this new learning rule is designed to approximate the outgoing radiance in direction ω from point x. Therefore, the value of the state-action pair $Q(x = S_t, \omega = A_t)$ is determined by the amount of radiance incident in direction ω from point x. An important detail which may be overlooked is that the substitution of $\gamma \cdot \pi(a|S_{t+1})$ for $f_s(\omega_k, y, -\omega) \cdot \cos(\theta_i)$ ensures that a trade off of long term rewards for more immediate rewards is made as $f_s(\omega_k, y, -\omega) \cdot \cos(\theta_i) \leq 1$. Meaning, the learning rule accurately accounts for a light paths loss of energy as reflects off surfaces in the scene.

The following lists what each term in the Expected Sarsa learning rule is matched to:

S_t	= 3D position in the scene, $x \in \mathbb{R}^3$
A_t	= Sampling a direction to continue the light path from location x, in direction ω
S_{t+1}	= 3D position of a light ray from reflected from x in direction ω, $y = h(x, \omega)$
R_{t+1}	= Emitted radiance from point y in direction $-\omega$, $L_e(y, -\omega)$
$\mathcal{A}$	= All direction in the hemisphere at x, oriented to the surface normal at x, Ω
$\gamma \cdot \pi(a\|S_{t+1})$	= BRDF and the cosine of the angle y, ω_i, $f_r(\omega_i, y, \omega) \cdot \cos(\theta_i)$
$Q(S_t, A_t)$	= Radiance incident on x from direction ω, $-L_i(x, \omega) = Q(x, \omega)$

$$Q(x,\omega) \leftarrow (1-\alpha)\cdot Q(x,\omega) + \alpha \cdot \left(L_e(y,-\omega) + \int_\Omega Q(y,\omega_i) f_s(\omega_i, y, -\omega)\cos(\theta_i) d\omega_i \right) \tag{2.26}$$

Finally, Monte Carlo integration with a uniform distribution for the probability density function can be used to approximate the integral in Equation 2.26. This converts the action space from continuous to n discrete angles and provides a numerical solution for approximating the outgoing radiance from x in direction ω, which is presented in Equation 2.27.

$$Q(x,\omega) \leftarrow (1-\alpha)\cdot Q(x,\omega) + \alpha \cdot \left(L_e(y,-\omega) + \frac{2\pi}{n} \sum_{k=1}^{n-1} Q(y,\omega_k) f_s(\omega_k, y, -\omega)\cos(\theta_k) \right) \tag{2.27}$$

The estimated outgoing radiance values evaluated using Equation 2.27 for a discrete set of angles around a given point x can then be converted in to a distribution. This distribution is accordingly named the radiance distribution for a point [24]. A good approximation of the radiance distribution at a point x will have a similar shape to the true function of outgoing radiance at point x, $L_o(x,\omega)$ $\forall\omega \in \Omega$. Therefore, by Monte Carlo importance sampling, sampling directions for light path construction from the learned radiance distribution and using it as the probability density function in the Monte Carlo approximation will significantly reduce the variance in the approximation. Leading to a significant reduction to a significant reduction in image noise.

Summary

The story up to this point can be summarised as follows; Monte Carlo integration is a numerical technique for approximating integral. It can be used to find an approximation of pixel values in path tracing by approximating the reflected radiance from a point in a given direction. Rather then uniformly sampling directions in light path construction for determining pixel values, a whole field is dedicated to importance sample these directions to reduce the variance in the Monte Carlo approximation. However, most traditional methods lack the ability to effectively reduce the number of zero-contribution light paths as they do not take into account visibility. Instead, reinforcement learning, specifically temporal difference learning techniques can be applied to learning the outgoing radiance from a point in a given direction, which are in turn used for importance sampling ray directions where the approximated radiance is high. To do this a link has been made between the rendering equation and the temporal difference learning rule Exepected Sarsa. This has opened up the opportunity to design and evaluate new algorithms for importance sampling directions in light path construction, specifically ones involving deep reinforcement learning.

Chapter 3

TD-Learning and Deep Reinforcement Learning for Importance Sampling Light Paths

Monte Carlo Integration, importance sampling, Monte Carlo path tracing, reinforcement learning and the link between temporal difference learning and light transport have all been introduced. So now it is finally time to combine all of these concepts to build path tracing algorithms for Importance sampling.

In this section I introduce two path tracers I have implemented to assess their ability in reducing image noise in Monte Carlo path tracing with a fixed number of sampled light paths per pixel. The first is based on a method introduced by Nvidia [16] which I refer to as the Expected Sarsa path tracer. The other I refer to as the Neural-Q path tracer which I have designed. Both of which progressively reduce the number of zero-contribution light paths sampled during rendering, reducing the variance in the Monte Carlo approximation of a pixels colour. Ultimately, reducing the noise in rendered images.

3.1 The expected Sarsa Path Tracer

The Expected Sarsa path tracer was first introduced by Nvidia [16] to apply the derived Expected Sarsa learning rule for learning the incident radiance in a given direction on a position, which was derived in Equation 2.27:

$$Q(x,\omega) \leftarrow (1-\alpha)\cdot Q(x,\omega) + \alpha \cdot \left(L_e(y,-\omega) + \frac{2\pi}{n}\sum_{k=1}^{n-1} Q(y,\omega_k) f_s(\omega_k, y, -\omega)\cos(\theta_k) \right)$$

I will now introduce my implementation of the algorithm which uses it to progressively reduce the number of zero-contribution light paths sampled in a path tracer, which can reduce noise in rendered images significantly.

3.1.1 The Irradiance Volume

The derived Expected Sarsa learning rule requires some sort of data-structure for looking up and updating the incident radiance on a point x from direction ω, which from here on I will refer to as a Q-value. Therefore, the main requirement of the data structure is that it has some way of representing a discrete set of angles (ω_k) in a hemisphere located at a position x and oriented to the surface normal at x. The Irradiance Volume data structure [24] meets this requirement.

Originally designed to be used for pre-computation of radiance values which are looked up at runtime to approximate global illumination, the Irradiance Volume data structure is essentially a discretized version of a hemisphere which is visually represented in Figure 3.1. The image shows the discrete sectors which make up a hemisphere, this was implemented by converting a 2D square grid into the 3D hemisphere shown, which is known as an adaptive quadrature. Where all sectors in the 2D grid have an equal area and a mapping introduced in [53] converts the 2D grid coordinates into a hemisphere made up of

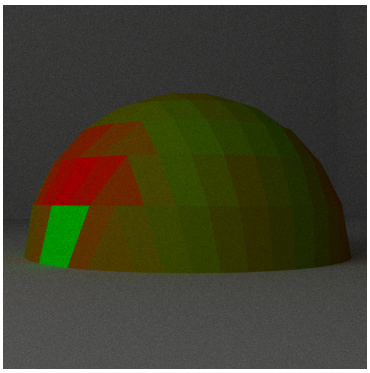
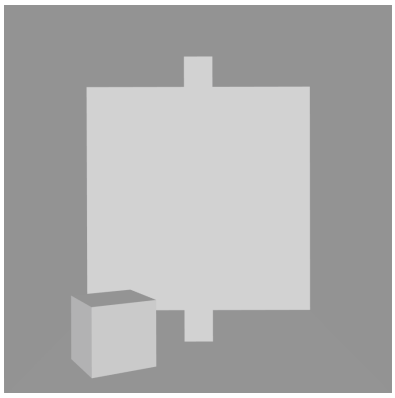
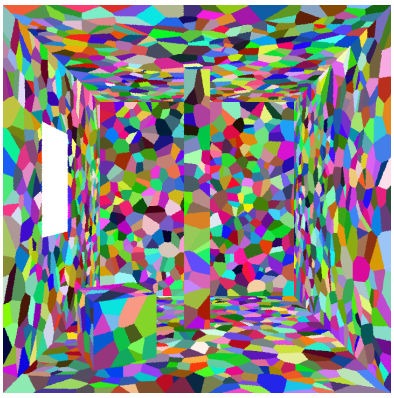


Figure 3.1: An Irradiance Volume. Each sector holds the incoming radiance $L_i(x, \omega_k)$, the more green a sector is the lower the stored radiance in that sector, the more red a sector is the higher the stored radiance in that sector.

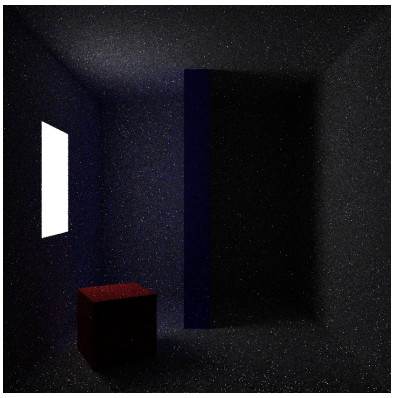
sectors in 3D space. The mapping ensures the hemisphere sectors remain equal to one another, meaning the discrete set of direction represented by the hemisphere are of equal angles apart from one another.



(a) Representation of the scenes geometry meshes



(b) Voronoi Plot of Irradiance Volume locations



(c) Expected Sarsa path tracer with 16 SPP

Figure 3.2: An example of discretizing location in the scene into Irradiance Volume locations. The geometry mesh (a) is used to uniformly sample Irradiance volume positions. Image (b) shows a voronoi plot for the Irradiance Volumes in the scene, where each pixel is coloured to the represent its closest Irradiance Volume, so each sector of colour in (b) represents a different Irradiance Volume location. Finally (c) gives a render using the Expected Sarsa path tracer based on Algorithm 2.

Each sector of the Irradiance Volume is then used to store the current approximation of incident radiance in the incident direction formed by the unit vector from centre of the sector to x. Therefore, an Irradiance Volume stores the incident radiance (Q-value) for a given position x (state) from each direction ω_k (action), for all sectors $\forall k = 1, ..., n$ in the hemisphere located at x. In order to store Q-values across the scene, Irradiance Volumes can be uniformly sampled over the scenes geometry as shown in Figure 3.2. Then to lookup a radiance/Q-value for a given position x in direction ω_k, a nearest neighbour search is performed to find the closest Irradiance volume to position x, then retrieve the Q-value from the sector at index k. Giving a lookup and update time of $O(\log n) + O(1) = O(\log n)$ when using the KD-Tree data structure for nearest neighbour search [9]. Lookup and update procedures from the Irradiance volumes in the scene is all that is needed to apply the Expected Sarsa learning rule in Equation 2.27. The entire collection of Irradiance Volumes in a scene can be thought of as a large 5D table ($x \in \mathbb{R}^3, \omega \in \mathbb{R}^2$) to store the current incident radiance approximation at each point. Expected Sarsa and all other TD-learning methods presented in section 2.3.4 are know as tabular methods, and I will refer to the table described as the Q-table for the scene.

At each Irradiance Volume the radiance distribution is stored. The radiance distribution for each

radiance volumes is a probability distribution over the n Q-values (incoming radiance estimates) stored by the radiance volume. As described in section 2.4, a direction to continue a light path in can be sampled proportionally to the radiance distribution of the closest radiance volume to the a light paths intersection point. This distribution is also used as the probability density function in the Monte Carlo path tracing colour estimate for a pixel (Equation 2.13). Due to this, as the radiance estimate held in each of the n sectors of every radiance volume improves with the number of rendered frames, the radiance distribution will come closer a normalized version of the true function describing the incoming radiance on point x. Therefore, the variance in the colour estimate for a light path through a given pixel will reduce for the same number of samples used, progressively reducing image noise.

3.1.2 Expected Sarsa Path Tracing

The Expected Sarsa based path tracing algorithm is very similar to the original forward path tracer introduced in Algorithm 1. The algorithm learns online, meaning after every rendered frame pixel values are likely to have a lower variance due to a reduction in the number of zero contribution light paths sampled. Initially, radiance volumes are sampled uniformly across the room with all Q-values initialised to a small constant proportional to the number of sectors on each hemisphere k. This encodes the assumption that initially the radiance in all directions from any given point in the room is equal, as initially there is no prior knowledge of any radiance values $Q(x, \omega)$. With the radiance volumes set up, every frame N sampled light paths are traced through each pixel from the camera and into the scene to find its average colour estimate using Algorithm 2. Algorithm 2 updates the radiance distribution stored in by each Irradiance Volume after every rendered frame using the updated $Q(x, \omega_k)$ values stored. Meaning the next frame will importance sample from the new updated radiance distribution estimate held by every Irradiance Volume. The three additions to this algorithm from the forward path tracer in Algorithm 1, are as follows:

Addition 1: Sample directions proportional to closest radiance distribution

The direction to continue the light path in is sampled proportional to the Q-values stored in the closest Irradiance Volume to position y by using the stored radiance distribution in the closest Irradiance Volume. Inverse transform sampling [17] is used to sample a direction from the stored radiance distribution. Inverse transform sampling is where a random number $r \in [0, 1]$ is sampled, then the largest number x from the domain of the cumulative distribution $P(X)$ is returned where $P(-\infty < X < x) \leq r$.

Addition 2: Update stored Q-values

Once the ray has intersected with a position in the scene y from a position x, update the radiance estimate $Q(x, \omega)$ using the Expected Sarsa learning rule derived in Equation 2.27. This is based on the radiance emitted from y in direction $-\omega$ and the outgoing radiance estimate in direction $-\omega$ from point y described by the summation. The summation involves summing all Q-values for the closest radiance volume to position y. Each of which are multiplied by BRDF of the surface at y, as well as the cosine of the angle between the sector direction for the Q-value (ω_k) and the surface normal y.

Addition 3: Update Irradiance Volume Distributions

Update every radiance volumes radiance distribution in the scene by normalizing the radiance volumes updated $Q(x, \omega_k)$ values.

Algorithm 2: Expected Sarsa path tracer pseudo code following Nvidia's method in [16]. Given a camera position, scene geometry, this algorithm will render a single image using a tabular Expected Sarsa approach to progressively reduce image noise. Where N is the pre-specified number of sampled light paths per pixel.

Function *renderImage(camera, scene)*
  for $i = 1$ **to** N **do**
    for p **in** *camera.screen* **do**
      $ray \leftarrow$ initializeRay$(p, camera)$
      for $j = 1$ **to** ∞ **do**
        $(y, \mathbf{n}, L_e) \leftarrow$ closestIntersection$(ray, scene)$
        if $j > 1$ **then**
          /* Addition (1) */
          $(\omega_i, \rho_i, f_s) \leftarrow$ sampleRayDirFromClosestRadianceDistribution(y)
          /* Addition (2) */
          $Q(ray.x, ray.\omega) \leftarrow (1 - \alpha(ray.x, ray.\omega)) \cdot Q(ray.x, ray.\omega) + \alpha(ray.x, ray.\omega) \cdot \left(L_e + \frac{2\pi}{n} \sum_{k=1}^{n-1} Q(y, \omega_k) f_s(\omega_k, y, -ray.\omega) \cdot (\omega_k \cdot \mathbf{n})\right)$
        end
        if *noIntersection*(y) **or** *areaLightIntersection*(y) **then**
          $ray.throughput \leftarrow ray.throughput \cdot L_e$
          updatePixelColourEstimate$(p, ray.throughput)$
          break
        end
        $ray.throughput \leftarrow ray.throughtput \cdot f_s \cdot (\omega_i \cdot \mathbf{n})/\rho_i$
        $ray \leftarrow (y, \omega_i)$
      end
    end
  end
  /* Addition (3) */
  updateIrradianceVolumeDistributions()
end

Monte Carlo Integration

Due to the importance sampling from addition (2), the probability density function over all possible sampled directions Ω at location x is no longer uniform. Instead it is equal to the normalized Q-values for the closest Irradiance Volume to the point x, which is the stored radiance distribution. Therefore the evaluated probability density function ρ_i at intersection point x, at incident direction ω_i must also be evaluated using the radiance distribution to correctly apply Monte Carlo Integration. To do so I have derived an Equation 3.1 to determine the value ρ_i.

$$\rho_i = Q_p(x, \omega_k) \cdot n \cdot \frac{1}{2\pi} = \frac{Q_p(x, \omega_k) \cdot n}{2\pi} \tag{3.1}$$

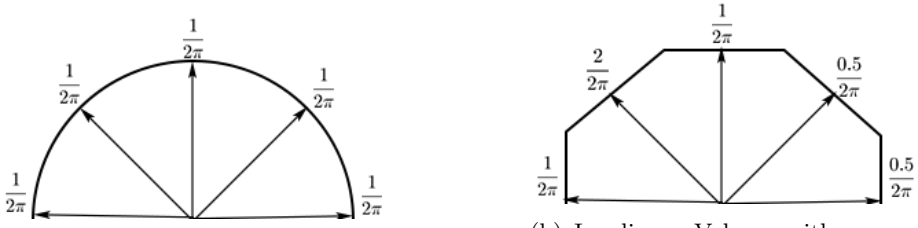
Where:

$Q_p(x, \omega_k)$ = Normalized Q-value from the Irradiance Volume closest to x at sector k
n = Total number of sectors in an Irradiance Volume

The reasoning behind this value ρ_i is that $\frac{1}{2\pi}$ represents the probability density function (*pdf*) evaluated at any point when directions are randomly sampled in the hemisphere Ω. However, the Irradiance Volume splits the hemisphere into discrete sectors, but each sector represents a continuous set of angles. Therefore if the probability of sampling a ray in each sector were constant c, $Q_p(x, \omega_k) = c \; \forall k < n$, the *pdf* would remain constant:

$$\frac{Q_p(x, \omega_k) * n}{2\pi} = \frac{1}{2\pi}$$

However, the approximated radiance may vary across sectors, causing the associated *pdf* to vary across sector due to $Q_p(x, \omega_k)$. This was not the case previously prior to importance sampling, where direction were sampled randomly over a unit hemisphere making a the *pdf* is uniform. The diagram in Figure 3.3 highlights these differences.



(a) Hemisphere with a uniform *pdf*

(b) Irradiance Volume with a non-uniform *pdf*

Figure 3.3: A 2 dimension view of a subset of values from two probability density functions (*pdf*). One for a unit hemisphere (left) with a uniform *pdf*. One for an Irradiance Volume (right) with non-uniform pdf. Where the arrows represent sampled directions and the values at the end are the evaluated *pdf* values for each direction.

Consistency

It is important for the modified path tracing algorithm to converge to ensure consistency, meaning all introduced artefacts such as image noise are guarenteed to vanish over time [16]. A decaying learning rate for α can be used to do so, see Equation 3.2. Where $\mathrm{i}(rv, \omega_k)$ is the number of times the rendering algorithm has updated the Q-value of the Irradiance Volume i for sector k representing angle ω_k.

$$\alpha(x, \omega) = \frac{1}{1 + \text{visits}(x, \omega_k)} \tag{3.2}$$

Now, while $Q(x, \omega_k)$ does converge, it does not necessarily converge on the true radiance $q_*(x, \omega_k)$. This is due to the discretization of the action/direction into sectors which make up a hemisphere. If the number of sectors was infinite then the algorithm would converge on the true radiance by the law of large number applied to Equation 2.27. Clearly this is not possible, but later I discuss how increasing the number of sectors on the Irradiance Volume affects the number of zero-contribution light paths. Another issue is Q-values will not be visited the same number of times during rendering. For example Irradiance Volumes located in places which are in the cameras view will be visited far more, so images rendered of the scene which have been in the cameras view for the longest are likely to have the lowest variance in pixel colours. This is a problem, as parts of the scene may look particularly noisy compared to others as the camera begins to move round the scene.

3.2 The Neural-Q Path Tracer

Up to now I have only spoke of TD-learning techniques which involve use a tabular approach to approximate the the optimal value function. However, it is possible to instead use a Neural Network as a non-linear function approximater for the optimal value function [59]. Following this I introduce an artificial neural network architecture capable of learning the incident radiance on any point in a scene from a discrete set of angles. I then propose the Neural-Q path tracing algorithm I have designed and implemented which uses the neural networks approximations for importance sampling direction to continue light paths in.

After, I take a slight detour to review the new materials that have been recently published on neural networks for importance sampling in Monte Carlo path tracing. The materials that focus on neural importance sampling for light path construction were published during the execution of the project, so it is important to find out where my Neural Q-learning algorithm sits compared to these new state of the art methods.

3.2.1 Introduction to Deep Reinforcement Learning

Value Function Approximation

Observe the optimal value function introduced in section 2.3.3, Equation 2.17:

$$q_*(s,a) = \max_\pi q_\pi(s,a)$$

Recall this is simply a function which given a state and action, outputs a scalar representing the value of that state action pair. Therefore, rather then approximating it using a tabular form, instead one can learn the a functions parametrized functional form with a weight vector $\theta = (\theta_0, ..., \theta_n)$ where $\theta_i \in \mathbb{R}$. This turns the task of approximating the value function into a function approximation problem, for which there are many possible methodologies. To name some, Coarse coding, Decision Trees, Nearest Neighbour, Fourier basis, and Artificial Neural Networks (ANNs) [56]. Function approximators other than ANNs have been successful for a range of reinforcement learning problems, whilst maintaining both data and computational efficiency [55, 32, 57]. However, I will be using an ANN for value function approximation due to its capabilities of learning a non-linear functions, and its performance in the presence of a large amount of training data [34]. The exact reasons these benefits are capitalized on will become more apparent later. By using a ANN for function approximation, the technique is now known as Deep Reinforcement Learning.

Stochastic Gradient Descent

The goal of the artificial neural network is to learn the value of the function parameters θ such that the functions loss over all possible state-action pair inputs is minimised. In the case of ANNs the parameters θ are the weights for the connections between neurons. For stochastic gradient descent a differentiable loss function which takes parameter vector θ as input and outputs a scalar loss value is required. The method from here on is to move the parameter values θ in the direction of the negative gradient to minimise the loss function:

$$\triangle\theta = -\frac{1}{2}\alpha\nabla_\theta J(\theta)$$

Where α is the step-size parameter. An example loss function for approximating the optimal value function q_π is given in Equation 3.3.

$$J(\theta) = \sum_{s\in\mathcal{S}}\sum_{a\in\mathcal{A}}(q_\pi(s,a) - q_\theta(s,a))^2 \tag{3.3}$$

Where:

$J(\theta)$ = The loss value function for the current parameter values θ
$q_\pi(s,a)$ = The value function under policy θ for the state-action pair
$q_\theta(s,a)$ = Current approximation of the value function under policy θ for the state-action pair

If plenty of data was available for $q_\pi(s,a)$ for many state actions pairs (s,a), it would be possible to train an ANN to approximate the optimal value function q_π. By simply running a forward pass on the

ANN to compute $q_\theta(s, a)$ for a given state-action pair as input, then calculating the loss $J(\theta)$ using the ground truth $q_\pi(s, a)$. Then, by using the backpropagation algorithm to calculate the partial derivatives w.r.t the loss, the parameters values θ can be updated using an optimizer such as Adam. The issue is q_π is unknown and no training data is initially available for the training procedure just described. Instead, deep reinforcement learning uses online training procedures closely resembling the TD-learning methods introduced in section 2.3.4 for function approximation.

Bootstrapping

Following TD-learning, as the optimal value function q_* (or q_π in the previous section) is not available, it is possible to instead bootstrap using the current estimate of the value function [59]. Recall that bootstrapping for the value function is when the updated current estimate of the optimal value function for a state-action pair is partially based on new experience data, and the current estimated value of the next selected state-action pair when following policy π. An example of this is the off-policy method Q-learning presented in section 2.3.4, Equation **??**. Equation 3.4 gives what is known as the TD error (δ_t) for Q-learning at time step t. It is called an error as it gives the difference between the current estimate $Q(S_t, A_t)$ and the better estimate $R_{t+1} + \gamma \max_a Q(S_{t+1}, a)$. Notice that the TD error is an estimate that has been made at time step t, meaning the error depends on the next state and the next reward which can both change across time steps.

$$\delta_t = \Big(R_{t+1} + \gamma \max_a Q(S_{t+1}, a)\Big) - Q(S_t, A_t) \tag{3.4}$$

It is this TD error that can be used to form the loss function for an ANN to approximate the value function, as shown in Equation 3.5. The gradient of this loss function can be calculated by Equation 3.6 which is used with an optimizer for updating the parameters θ during learning. This is the only deep reinforcement learning rule I shall be using from here on for the Neural-Q path tracer.

$$J(\theta) = \Big(R_{t+1} + \gamma \Big[\max_a \hat{q}_\theta(S_{t+1}, a)\Big]\Big) - \hat{q}_\theta(S_t, A_t) \tag{3.5}$$

$$\nabla_\theta J(\theta) = \Big(\Big(R_{t+1} + \gamma \Big[\max_a \hat{q}_\theta(S_{t+1}, a)\Big]\Big) - \hat{q}_\theta(S_t, A_t)\Big) \nabla_\theta \hat{q}_\theta(S_t, A_t) \tag{3.6}$$

Where:

∇_θ = Gradient w.r.t θ
$\hat{q}_\theta$ = The current approximation of the value function given by an ANN
$[.]$ = Stop gradient, meaning the value is taken as just a scalar input during backpropagation

Unfortunately due to the use of function approximators for TD-learning methods, the methods are no longer proven to converge on the optimal policy q_*. That said, in general these methods perform well in practice for appropriate use cases [35, 36], and generally learn faster than the other option of Monte Carlo Reinforcement Learning with function approximation [56].

3.2.2 Deep Reinforcement Learning Motivation

Like tabular TD-learning methods, I attempt to a deep reinforcement learning method to approximate the incident radiance from a set of discrete directions ω_k where $k = 0, ..., n$ from point x in the scene $(L_o(x, \omega_k))$, where n is the number of discrete directions in the hemisphere. Then, when sampling direction to continue light paths during path tracing, the incident radiance estimate from each discretized direction is normalized, forming a distribution known as the radiance distribution, see section **??**. A good approximation of the radiance distribution at a point can be used in importance sampling directions for light path construction during Monte Carlo path tracing in order to reduce image noise (see section 2.2.2).

Unlike the Expected Sarsa tabular approach, function approximation has the ability to generalize the value over state-action pairs. In other words, the tabular approach requires a single entry for every possible state-action. So, if an update is made to a state-action pair, it affects that state-action pair value alone. In the case where there are many states (potentially infinitely many) state-action pairs may not be updated often due to the number of them. This means they will be updated infrequently, making learning slow. It may not even be possible to store such a number of state-action pairs. Using an ANN or any function approximator allows one to model a potentially infinite state space and generalize for the valuation of for unseen state-action pairs, which applies well to rendering as the number of unique

positions in the scene is infinite. However, this comes at the cost of updating weights in the ANN can affect the valuation of multiple states, making it impossible in practice to predict the optimal value for every state-action pair, unlike the tabular case. This causes proofs for convergence on the optimal value function, $q_*(s, a)$ to break down.

Previously, I highlighted that I chose an ANN as the function approximator for the Neural-Q path tracer due to its ability to model non-linear functions well in the presence of a large amount of training data. When approximating the incident radiance $L_i(x, \omega)$ for all possible positions in the scene, the radiance distribution at any point in the scene can be a non-linear function due to the way light paths randomly reflect off complex geometry in a scene. Therefore, learning the radiance distribution for all points in a arbitrary scene requires a non-linear function approximator. In terms of training data, the rendering process samples light paths to approximate radiance which can be used for training. This means as much training data as needed can be generated during the rendering process.

The range of improvements to the approximation of incoming radiance that can be potentially made by using ANNs makes it a clear area to investigate. However, this is all conditioned on if it is possible for an ANN to learn the radiance distribution for any point in an arbitrary scene.

3.2.3 Deep Q-learning for Light Transport

In order to use deep reinforcement learning to learn the incident radiance on a point from a given direction in the scene, I must first derive a loss function for which the neural network will be trained with in a similar way done for the Expected Sarsa learning rule in section 2.4. However, instead of using Expected Sarsa, I find a loss function using the deep Q-learning rule introduced in Equation 3.5. This choice was made primarily due to its proven success when used to approximate the optimal value function for a variety of Atari games [36].

Once again, the rendering equation from section 2.2.1 states the radiance in an outgoing direction ω from point x is equivalent to the emitted radiance in the direction plus the reflected radiance in the direction:

$$L_o(x, \omega) = L_e(x, \omega) + \int_\Omega L_i(h(x, \omega_i), -\omega_i) \cdot f_r(\omega_i, x, \omega) \cdot \cos(\theta_i) d\omega_i$$

By matching terms and adapting the rendering equation to the Deep Q-learning loss function in Equation 3.5, the loss function for training an ANN to learn the incoming radiance in a direction ω on a point x can be found, see Equation 3.7.

$$\triangle \hat{q_\theta}(x, \omega) = \left(L_e(y, -\omega) + \left[\max_{\omega_i} (\hat{q_\theta}(y, \omega_i) f_s(\omega_i, y, \omega)(\omega_i \cdot \mathbf{n})) \right] \right) - \hat{q_\theta}(x, \omega) \tag{3.7}$$

Where:

$\triangle \hat{q_\theta}(x, \omega)$ = The loss/error of the ANNs approximation of $\hat{q_\theta}(x, \omega)$
θ = Current parameter values of the ANN
ω_i = The direction where the incident radiance is highest on y
$(\omega_I \cdot \mathbf{n})$ = Equivalent to the cosine of the angle between the normal $\mathbf{n}$ and direction vector ω_i
$(\cdot)$ = Denotes the dot product

3.2.4 Artificial Neural Network Architecture

With the loss function defined in Equation 3.7 for learning the incident radiance on a point from a given direction, a suitable ANN architecture must be developed for approximating the $\hat{q_\theta}(x, \omega)$. At first one might the ANN would take a single 3D position x and an incident direction ω as input for a forward pass to calculate the approximated valuation under parameters θ. This way the ANN could take any arbitrary incident direction ω to calculate the incident radiance on position x. However, when the radiance needs to be estimated for each discrete direction ω_k in the hemisphere around a position x for all n directions, n forward passes must be made through the ANN must be made. A single light path may include hundreds of reflections before intersecting with a light source, meaning potentially thousands of forwards passes would need to be evaluated for a single light path. To be conservative, imagine every light path reflected 30 times before intersecting with a light source, if we only sample 16 light paths per pixel for a $512x512$

image. The total number of forward passes for using only 16 different possible directions to sample a light path in every time it is reflected is over a billion.

To avoid this situation, I followed the technique proposed for learning to play Atari games in [36]. Which is in the context of reinforcement learning to give the ANN as input the agents state, then a forward pass of the network gives the value (Q-value) of each state-action pair for the input state. Applying this to incident radiance where the input state is the position x, a forward pass computes the radiance incident from the set of discrete directions $\omega_k \ \forall k = 0, ..., n$. In other words, a single forward pass of the ANN gives all the required information needed to importance sample a direction to continue the light path in.

With the current process proposed only a single 3D point will be passed into the ANN to infer the radiance in directions $\omega_k \ \forall k = 0, ..., n$. The 3D position given is in the world coordinate system of the scene, hence it gives no information regarding where that point is relative to the geometry in the scene. In terms of reinforcement learning, the state is said to be only partially observable [56]. Instead, I found that in order for the network to learning the radiance from incident directions ω_k, the input state should instead be the coordinates of all vertices in the scene in a coordinate system relative to the position the incident radiance is being estimated at. Formally, for any position 3D $x \in \mathbb{R}^3$ in the scene with a set of vertices $\mathbf{v} = (v_0, v_1, ..., v_m)$ where $v_i \in \mathbb{R}^3 \ \forall v_i \in \mathbf{v}$, an input vector $\mathbf{v^x}$ was formed:

$$\mathbf{v^x} = (v_0^x, v_1^x, ..., v_m^x)$$
$$\text{where: } v_i^x = v_i - x \quad \forall i = 0, ..., m$$

This decision was inspired by [36], where a state in a game of Atari is represented by the raw image of the game. This encodes information regarding where the play is relative to objects around it in 2D, whereas $\mathbf{v^x}$ encodes where the current location x is to all objects around it in 3D. The full process of approximating the incident radiance on x from discrete directions ω_k is illustrated in Figure **??**.

As shown in the illustration in Figure **??**, the ANN consists of an input layer consisting of z neurons, where z is the the total number of vertices in the scene. The input is then all verticesin the scene converted into a coordinate system around the intersection position x, $\mathbf{v^x}$. Then there are 2 hidden fully connected layers, where each hidden layer is followed by a rectifier of non-linearity activation function (ReLU) [40]. The output layer consists of a neuron to output the approximated radiance in every direction of the discretized hemisphere around the intersection point x. The choice to use only two fully connected hidden layers was made to reduce time spent on computing forward passes and training the network during the rendering of an image. Whilst still having two hidden layers allows the ANN to represent an arbitrary decision boundary for approximating incident radiance at any given position which is a 5D function $L_i(x, \omega)$, where $x \in \mathbb{R}^3$ and $\omega \in \mathbb{R}^2$. However, for scenes with more complex geometry than those experimented on may require more hidden layers and/or neurons in hidden layers in order to accurately approximate $L_i(x, \omega)$ [50].

Training the ANN is where the main difference arises compared to the supervised learning case. Recall from section 3.2.1, in the reinforcement learning case there is no large training data set available for learning the optimal value function directly. Hence, bootstrapping is used in TD-learning methods in order to learn online by continually updating the estimate of the optimal value function as data is received. This is otherwise known as learning from experience [56]. The loss function in Equation 3.7 does exactly that, where the loss is described in terms of some immediate reward which is the emitted radiance L_e term, and partially on the current estimate of radiance incident from direction ω:

$$\text{TD Error: } \triangle\hat{q_\theta}(x, \omega) = \left(L_e(y, -\omega) + \left[\max_{\omega_i} (\hat{q_\theta}(y, \omega_i) f_s(\omega_i, y, \omega)(\omega_i \cdot \mathbf{n})) \right] \right) - \hat{q_\theta}(x, \omega)$$

So, to compute this loss an initial forward pass on the network must be made to determine the value of $\hat{q_\theta}(x, \omega)$, then a second pass must be made to calculate $max_{\omega_i} \hat{q_\theta}(y, \omega_i)$. All other terms are a result of experience and are calculated in the path tracing algorithm. After the loss is calculated backpropagation is used to compute the partial derivative of the loss with w.r.t to each parameter. Then another pass can be made through the network to update the networks weights using the partial derivatives calculated in the previous step with an optimizer, in this case I use the Adam optimizer. While there is no guarantee,

in practice ver many iterations of this learning procedure, the ANNs estimate of incident radiance for a discrete set of directions ω_k for any point x in the scene should move towards a local optimum [59].

3.2.5 Neural-Q Path Tracing Algorithm

With an ANN and a method for training it to learn the the incident radiance from n discrete directions on a point detailed, a modified Monte Carlo path tracing algorithm which uses this ANNs predictions for importance sampling directions to continue light paths must be introduced. Algorithm 3 renders a single image using N sampled light paths per pixel to compute the colour estimate of each pixel by Monte Carlo path tracing. Similarly to the Expected Sarsa path tracer detailed in Algorithm 2, the Neural-Q path tracer estimates the radiance distribution at the intersection point of a light path and samples a direction to continue the light path in proportional to this distribution. This radiance distribution is also used as the probability density function in the Monte Carlo estimate of the rendering equation 2.13 to reduce the variance in the approximation of the outgoing radiance L_o, leading to a reduction in image noise. The algorithm also trains online, meaning it progressively reduces noise in the image as simulates more light paths. Once again the ρ_i term is the probability density function over the hemisphere of directions Ω at intersection point x to continue a light path in, evaluated for the sampled direction ω_i. This can be calculated using Equation 3.1 in section 3.1.2. There are four additions to this algorithm compared to the default forward path tracing algorithm 1:

Addition 1: Minibatching

In order to yield smoother sample gradient for training, a minibatching method is used where the gradient information for light paths is accumulated over a batch to train the network with. This was proven to be succesful for a range of physical control tasks in[35]. While Algorithm 3 iterates through the batches to clearly represent the steps the algorithm takes, in practice all computation with the batch loops are done in parallel.

Addition 2: Sample direction using decaying ϵ-*greedy* policy

The Neural-Q path tracer follows a decaying ϵ-*greedy* policy to either sample a ray direction proportional to the estimated radiance distribution at the intersection position. Or, randomly sample one of the discrete directions in the discretized hemisphere centred at the intersection point. This choice is made depending on the value of ϵ. Where $\epsilon \in [0, 1]$, a random number $r \in [0, 1]$ is sampled. If $r > \epsilon$ then the so called *greedy* strategy is chosen. This is where a direction is sampled proportional to the radiance distribution formed by normalizing the Q-values produced by a forward pass on the network using $\mathbf{v^x}$ as input, where x is the current intersection position of the light path. Otherwise, a random direction ω_k is selected from the discrete set directions ω_k for $i = 1, ..., n$ for the adaptive quadrature at the point of intersection.

Addition 3: Training the ANN

This modification trains the ANN to improve its approximation of the incident radiance at any point x in the scene for the set of discrete angle ω_k in the adaptive quadrature centred at x. To do so, the algorithm computes the loss function in 3.7, by running a forward pass on the ANN to get the approximated incident radiance in the discrete directions ω_k $\forall k = 1, ..., n$ for both the position x, as well as the next intersected position y. The loss is then used to train the network using the process described in section 3.2.4. The policy followed for selecting a direction to continue a light path in is different to that of selecting a direction Q-value for training with in $q_t\hat{heta}(y, \omega)$ for bootstrapping. This is instead chosen according to the direction with the highest incident radiance from y as shown in 3.7. This is the reason why deep-Q learning is an off-policy algorithm.

Whilst a replay buffer was not used in the Neural-Q path tracer, what it hopes to achieve is partially covered by the Neural-Q path tracing algorithm. A replay buffer was originally introduced for deep Q-learning in [36] for training an AI agent to play a variety of Atari games by learning a good approximation of the optimal value function using deep Q-learning. Where the replay buffer stores transitions between from state-action pairs and the corresponding reward received $e_t = (S_t, A_t, R_t, S_{t+1})$. The replay buffer is then sampled from to make up part of the minibatch to train the network with. This is known as experience replay and is used to avoid giving the network highly correlated data to train on as a result

of learning online, taking one action in a particular states may affect the probability of selecting another action in the next state. Neural networks make the assumption that the training data is independently and identically distributed (i.i.d), which was not the case for consecutive states in Atari games. Therefore, by sampling from a replay buffer the variance in consecutive weight updates is reduced as they updates are no longer highly correlated with one another compared to using standard Q-learning [56]. Rather then storing transitions in a buffer to make up a minibatch for training, a batch of rays are continually traced and trained together. This avoids strong correlations between training iterations, reducing the variance in consecutive weight updates. However, it is likely there still is some correlation in the initial consecutive training iterations for light paths sampled from spatially local pixel positions in the image plane, so using some form of replay buffer like that suggested in [39] may further reduce the variance in consecutive weight updates.

Addition 4: Decaying decaying ϵ-*greedy*

As part of the decaying ϵ-*greedy* policy, the value of ϵ is decreased after 1 sampled light path is computed through every pixel in the image. Meaning, according to Algorithm 3, this represents a single epoch. This is a reasonable point to decay epsilon as for an average light path length of 30 for each sampled pixel in a $512x512$ image with a minibatch size of 1024, approximately $7,680$ training updates are made on the network. As discussed in section **??**, the decaying ϵ-*greedy* policy means the makes sure the path tracer at first prioritises exploration of which direction contribute the most radiance. As training progresses, the path tracers policy alters more towards exploiting by sampling directions to continue light paths in based on the current estimated radiance distribution at their intersections points. This behaviour is desirable, as by focusing on exploitation to early, the path tracer may not sample seemingly unfavourable directions in the short term which after more reflections off surfaces lead to a large amount of incident radiance to

the point.

Algorithm 3: Neural-Q forward path tracer. Given a camera position, scene geometry, epsilon and epsilon decay, this algorithm will render a single image using deep Q-learning loss to progressively reduce image noise. Where N is the pre-specified number of sampled light paths per pixel.

Function *renderImage(camera, scene, decay, ϵ)*
  ANN = loadANN()
  for $i = 1$ **to** N **do**
    /* Addition (1) */
    for $b = 1$ **to** *Batches* **do**
      for $k = 1$ **to** *BatchSize* **do**
        $p \leftarrow$ getPixel(b,k)
        $ray \leftarrow$ initializeRay$(p, camera)$
        for $j = 1$ **to** ∞ **do**
          $(y, \mathbf{n}, L_e) \leftarrow$ closestIntersection$(ray, scene)$
          if $j > 1$ **then**
            /* Addition (2) */
            $(\omega_i, \rho_i, f_s) \leftarrow$ sampleRayDirEpsilonGreedy(ϵ, y)
            /* Addition (3) */
            $\hat{q_\theta}x \leftarrow ANN.getQValue(ray.x, ray.\omega, scene)$
            $\hat{q_\theta}y \leftarrow ANN.getMaxQValue(y, scene)$
            $\triangle\hat{q_\theta}x \leftarrow L_e + (\hat{q_\theta}y \cdot f_s \cdot (\omega_i \cdot \mathbf{n})) - \hat{q_\theta}x$
            ANN.train$(\triangle\hat{q_\theta}x)$
          end
          if *noIntersection*(y) **or** *areaLightIntersection*(y) **then**
            $ray.throughput \leftarrow ray.throughput \cdot L_e$
            updatePixelColourEstimate$(p, ray.throughput)$
            break
          end
          $ray.throughput \leftarrow ray.throughtput \cdot f_s \cdot (\omega_i \cdot \mathbf{n})/\rho_i$
          $ray \leftarrow (y, \omega_i)$
        end
      end
    end
    /* Addition (4) */
    $\epsilon \leftarrow \epsilon - decay$
  end
end

Chapter 4

Critical Evaluation

Here, a range of techniques are used to evaluate the newly proposed Neural-Q path tracer against the Expected Sarsa path tracer presented in Chapter 3 on its ability to reduce noise in rendered images produced by Monte Carlo path tracing. The introduction of deep reinforcement learning for importance sampling light transport paths in Monte Carlo path tracing comes with a variety of benefits which are tested and discussed in detail. However, the methods viability in industry is currently found to be limited due to the time taken to evaluate a forward pass through an ANN.

4.1 Experimental Setup

A path tracing renderer was built from scratch which supported Algorithms 1, 2, 3 and was used to produce all rendered results seen in this thesis. The rendering engine used only OpenGL Mathematics library [1] for various operations that are common in the rendering pipeline, SDL2 [2] for displaying rendered images, and the Dynet neural network library [41] for the Neural-Q path tracer implementation. Algorithms 1, 2, 3 were all accelerated on an Nvidia GPU by using the CUDA Toolkit [42] to receive experimental results in an acceptable time. The reasoning for choosing Dynet over more commonly used neural networks libraries such as Tensorflow [3], was due to its ability to be easily compiled with the CUDA `nvcc` compiler and it had a well documented C++ API. This was a requirement for the Neural-Q path tracer as tracing light paths, ANN inference, and ANN training are all performed on a Nvidia GPU via C++ API calls. All results were produced using a machine with an Intel i5-8600K CPU, Nvidia 1070Ti GPU and 16GB of RAM installed.

I developed four different scenes using Maya [6] and created a custom object importer to import the scenes into my path tracer renderer. A reference image of all four scenes is given in Figure **??**, which have been rendered with 4096 sampled light rays per pixel using the default forward path tracing Algorithm 1. They are known as reference images as they have minimal visible noise, meaning the Monte Carlo approximations for each pixel's colour value has approximately converged. Due to path tracing accurately modelling physical light transport (see section 2.2), these images are the ideal case for which all other path tracing algorithms should aim to produce as closely as possible with as few samples as possible.

4.2 Assessing the reduction in image Noise for Monte Carlo Path Tracing

4.2.1 Quantifying the Reduction in Image Noise

To quantify the amount noise within images rendered by a default forward path tracer, the Expected Sarsa path tracer, and the Neural-Q path tracer, I will use the Mean Absolute Percentage Error (MAPE) given in Equation 4.1 [39].

$$M = \frac{1}{N} \sum_{i=0}^{N-1} \left| \frac{A_i - F_i}{A_i} \right| \tag{4.1}$$

Where:

N = The total number of pixels in the image
A_i = The ith pixel value in the reference image
F_i = The ith pixel value in the image whose noise is being quantified

The MAPE value can therefore be used to quantify the average difference between pixel values from an image rendered by each rendering technique with the same number of sampled light rays per pixel and the reference image. Therefore, a lower MAPE score is desirable for a rendered image as it means the image has a lower amount of noise.

4.2.2 A Closer Inspection of Pixel Error Values

By visual observation of the rendered images 4.1, the type of noise present in the Expected Sarsa and Neural-Q path tracer's renders is quite different. In particular, the noise resulting from the Expected Sarsa path tracer are pixels with very high RGB values compared to that of their neighbours, which are commonly referred to as 'fireflies' [13]. Whereas the noise present in the Neural-Q renders is more subtle, however its general presence can be clearly seen when comparing these renders to that of the reference image. In order to investigate this further I have provided histograms in Figure 4.2 for the frequency of average RGB pixel error values of the rendered image compared to the reference image. Essentially, the greater the average RGB error for a particular pixel, the higher the amount of noise present in the pixel value estimate.

4.2.3 Convergence for Learning Incident Radiance

- Path length reduction
- Training curve to show convergence time, can compare the number of zero contribution light paths as when this peaks the algorithm has converged
- 1SPP taking the max direction and outputting the image to see which has converged on where the light is

4.2.4 Hyper Parameter Tuning

- Hyperparams in expected sarsa include what the min of radiance distribution can be set to, the selection of the learning rate α, the memory used for the scene...
- Hyperparams for Neural-Q on the other hand are the epsilon start and epsilon decay

4.2.5 Reusing Trained ANN for Different Scenes

- Expected Sarsa cannot be reused for a slightly different scene whereas the Neural-Q path tracer can
- Show the Neural-Q path tracer ability to be reused

4.3 From Memory Bound to Compute Bound

Computational resources utilized by the path tracing algorithms introduced is a very important talking point when considering the potential to be integrated into renders used in industry such as [22, 12, 54]. In particular any bounds imposed on the algorithm from current available hardware are the most important to consider, as rendering algorithms should be designed to give as much power to artists as possible to create and render any arbitrary scene of their choice. A detailed investigation on both highly optimised implementations of the Expected Sarsa and Neural-Q path tracers performance using current available hardware warrants a detailed investigation in itself. Instead for completeness, this section presents a high level analysis of how both memory usage and compute power limits their performance.

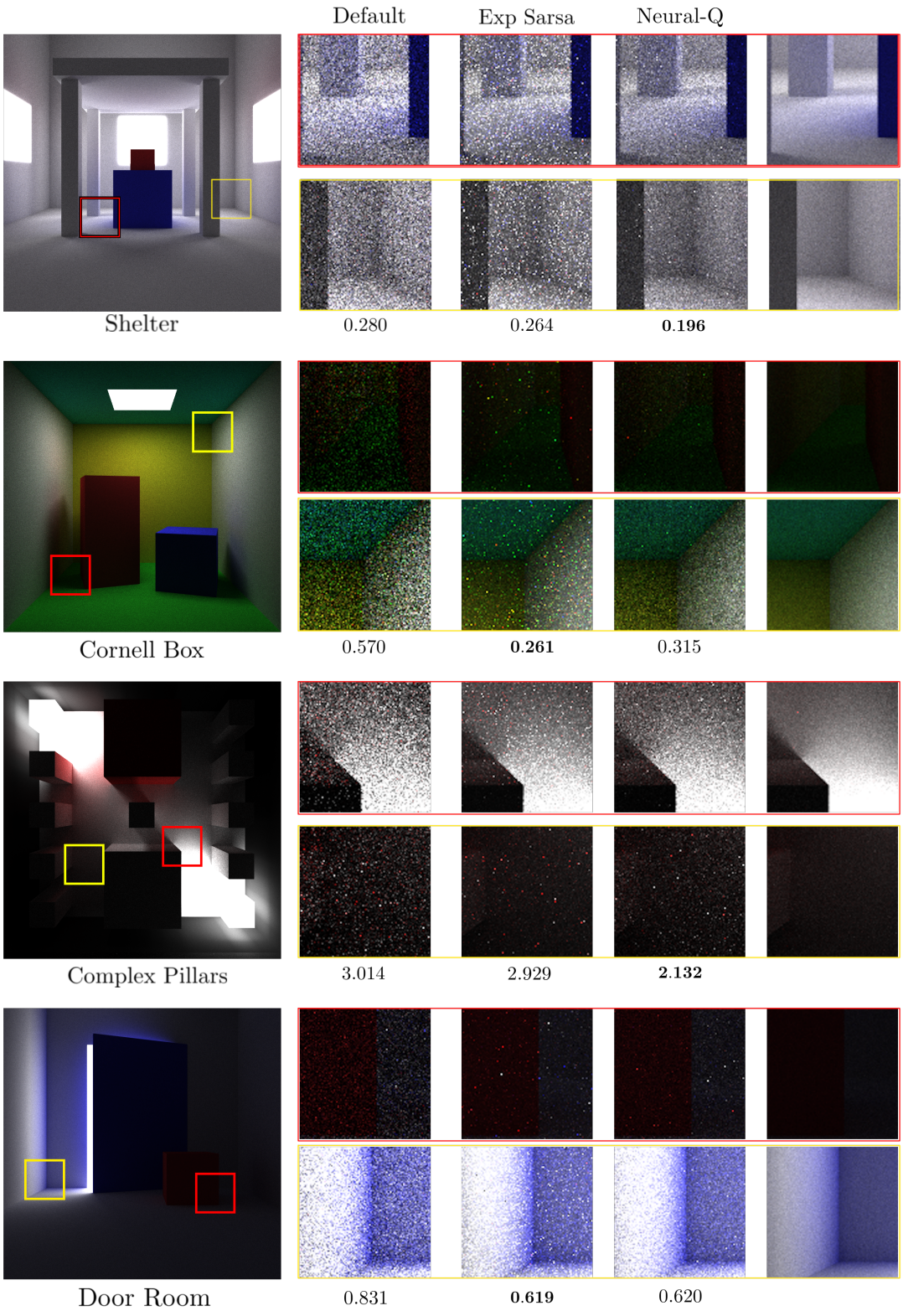


Figure 4.1: A comparison of the default forward path tracer, Expected Sarsa path tracer, and the Neural-Q path tracer of their image noise for four different rendered scenes. All scenes were rendered with 128 samples per pixel. The score under each column in an image row corresponds to the MAPE score for each path tracing algorithm for the particular scene.The Neural-Q and Expected Sarsa algorithms both used 144 equally spaced directions to estimate the radiance distribution on a given point. The Neural-Q path tracer used the network described in 3.2.4 for all fours scenes with a decaying ϵ-*greedy* policy start at $\epsilon = 1$ with a decay of $\delta = 0.05$ applied after every pixel in the image has had a light path sampled through it once. The Expected Sarsa path tracer used just enough Irradiance Volumes (which varied depending on the scene) to facilitate a significant reduction in image noise in all four renders.

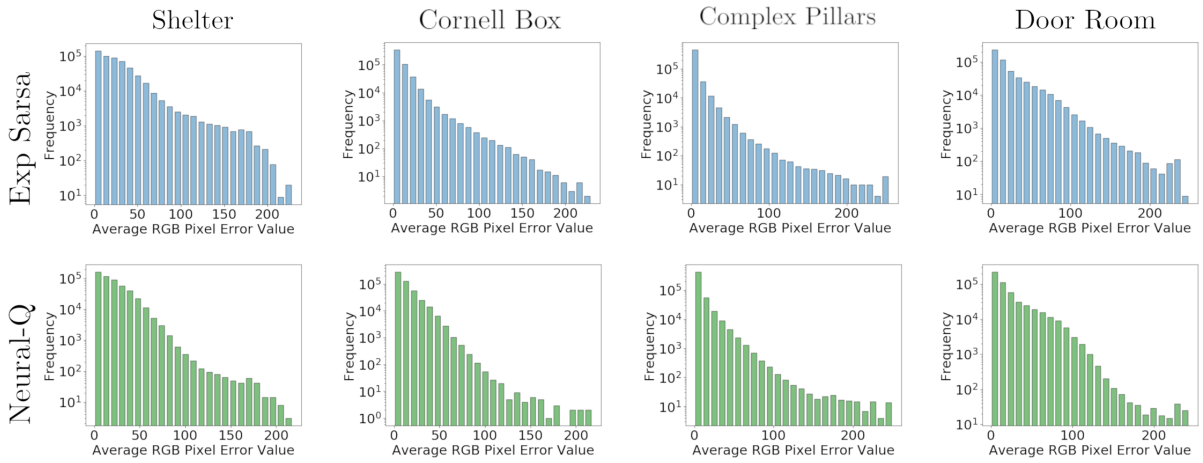


Figure 4.2: Histograms for the average RGB pixel error values for all four rendered scenes using both the Expected Sarsa path tracer and the Neural-Q path tracer. Where the average RGB pixel error value is the average difference in all RGB colour channels between a pixel in the reference image, and the corresponding pixel in the image rendered by either the Expected Sarsa or Neural-Q path tracers. The histograms are based on the rendered images presented in Figure 4.1.

4.3.1 Memory Usage

The default forward path tracer is compute bound, meaning the only way to make the algorithm faster is by providing more compute power. For example by parallelizing the rendering process using a GPU with a quicker processor clock speed. On the other hand, the performance of the Expected Sarsa algorithm for reducing image noise has been found to be bound by the amount of RAM the algorithm has available to it from the underlying hardware. This is due to the requirement of storing the Q-table in RAM which holds the approximated incident radiance values for a scene.

Expected Sarsa

The size of memory used by the Q-table is defined by the number of Irradiance Volumes sampled across the scenes geometry, as well as the number incident radiance values stored for the discrete directions ω_k $\forall k = 1, ..., n$ within each Irradiance Volume. I have found the higher the number of sampled Irradiance Volumes the more accurate the incident radiance approximation is for a given intersection point, as shown in Figure **??**. I have found this is due to the average distance from a light path intersection point to the closest Irradiance Volume decreases as the number of Irradiance Volumes sampled in the scene increases. This becomes clear by observing Figure **??**, where the Voronoi plot of the scene with a lower number of sampled Irradiance Volumes on average has a larger sector size, meaning the nearest neighbour search on average will have a higher distance to each Irradiance Volume compared to that of the scene with more Irradiance Volumes. Recall that a nearest neighbour search is used to estimate the incident radiance on a point based on the stored values in the closest Irradiance Volume, therefore the closer the Irradiance Volume used for this approximation the more accurate the approximation of incident radiance on the point will be.

A clear problem from this relationship is that if the underlying system does not have enough RAM, then the algorithm will not be able to significantly reduce the amount of noise in rendered images. Now, this may not be a problem for most small scenes with a polygon count ofaround 100, as common commodity graphics cards such as the NVIDIA 1070Ti GPU used for my experiments comes with 8GB of global memory (RAM), and NVIDIA claim they are able to fit the Q-table for a such scenes into only 2MB of memory to receive a significant reduction in noise [16]. However, films commonly render scenes with hundred of thousands, or even millions of polygons potentially making it impossible to store a Q-table large enough to significantly reduce image noise in RAM. For example, in Transformers Revenge of the Fallen, the robot 'Devastator' was made out of 11,716,127 polygons [51]. To present this memory scaling issue, Figure **??** presents a render of the Cornell Box scene which uses only xMB of memory to store the

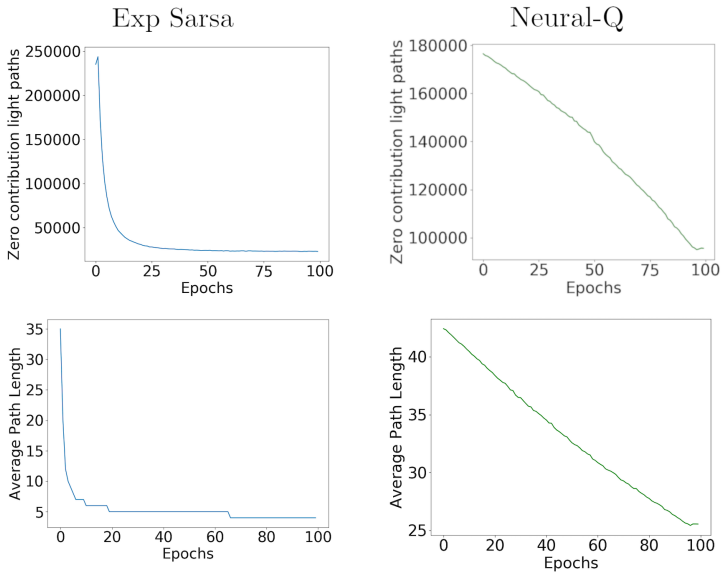


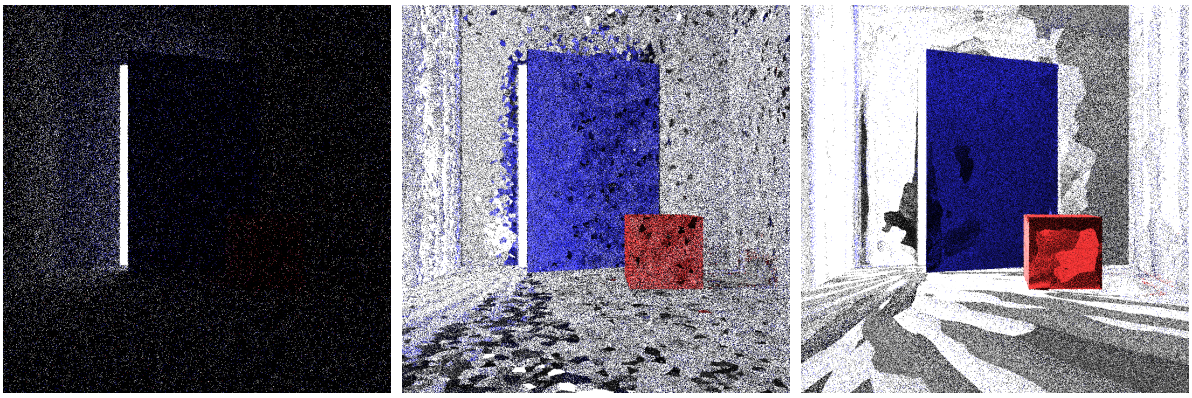
Figure 4.3: Training curves for the average path length and number of zero contribution light paths when rendering the Shelter scene for 100 epochs, using both the Expected Sarsa and Neural-Q path tracers. An epoch represents one sampled light ray through every pixel in the image. The average path length is the number reflections a light path takes before intersecting with a light source. A zero contribution light path is one which contributes almost zero colour to the final image.

Q-table. The artefacts present in the image are cause by vastly incorrect importance

Neural-Q

The Neural-Q path tracer on the other hand does not have exhibit the scaling in memory requirements as that of the Expected Sarsa path tracer. The is due to the Neural-Q path tracer only requiring more memory to store the incident radiance values for the current batch of light paths, which are used to importance sample directions for continuing the batch of light paths in. As well as, the amount of RAM required to store the ANN. The memory required to store the current batch of incident radiance values is controllable by the batch size parameter for the number of rays, therefore does not scale with the complexity of the scenes geometry. But a study from [50] suggests more layers may be required in ANNs to approximate the incident radiance $L_i(x, \omega)$ for scenes with more complex geometry, meaning more memory is required to store more weight parameters of the network. However, further experimentation must be done to investigate the scaling of memory usage to scenes with thousands of vertices for the two rendering methods, by using a more optimised rendering engine then the one I have built. The renders presented in Figure 4.1 were produced using a constant ANN architecture as specified in 3.2.4, which required only 30MB of memory to store the ANN as well as intermediate values computed during a forward pass. Whilst the Expected Sarsa path tracer required the rendered scenes with different geometry to have significantly different Q-table sizes to reduce rendered image noise. This is due to the ANNs ability to generalize the approximated incident radiance over the 5D space formed by a position $x \in \mathbb{R}^3$ and direction $\omega \in \mathbb{R}^2$. Where a single weight in the ANN has the capacity to contribute to the approximated radiance of many positions and directions in the scene. Hence, an ANN with 2 hidden layers was able to make a good approximation of the incident radiance for any (x, ω) for all tested scenes. The Q-table on the other hand holds an individual value for each incident radiance estimate (x, ω), therefore the storage requirements rose for the scenes rendered with more complex geometry. For example, the Q-table required for the Expected Sarsa rendering of the large archway scene without any artefacts used 272MB of data storage, 9X more than that of the Neural-Q path tracer.

Moreover, the higher the number of discrete directions$n$ held in each Irradiance Volume, the more accurate the estimated radiance distribution at the given intersection point will be, see Figure **??**. This is because the true radiance distribution at any point in the scene is continuous, so the more samples used in the Monte Carlo approximation of this probability density function (see section 2.4), the lower



(a) Default (b) Expected Sarsa (c) Neural-Q

Figure 4.4: The Door Room scene rendered using only 1 sampled light path per pixel with the default path tracer, a trained Expected Sarsa path tracer, and a trained Neural-Q path tracer. The light path through each pixel using the default path tracer is constructed according to Algorithm **??**. Whereas, the single sample for each pixel in the Expected Sarsa and Neural-Q renders is constructed by reflecting the light path in the direction of highest estimated radiance upon every intersection point, until the light is intersected.

the approximation error due the law of large number in Equation 2.5.

4.3.2 Parallel Processing

For high quality, high frame rate rendering of scenes, rendering algorithms in general become very computationally demanding [15]. In order to facilitate this high level of performance, parallel computing must be used to distribute the computational workload. This pattern follows for both scanline renderers and ray-tracing renderers (which path tracing falls into) [4, 20]. The default path tracing algorithm in particular is an embarrassingly parallel algorithm [21], as the task of finding the colour estimate of a single light path can be performed independently by a single process without any communication to other processes. A simple example of this is given in [5], whereby using GPU the process of rendering an image using an algorithm which closely resembles path tracing is split among many cores on a GPU. In industry path tracing algorithms take advantage of parallelism to an unprecedented scale, such as Disney's supercomputer for rendering full-length films using the Hyperion renderer [54]. Therefore, for any modified path tracing algorithm to gain industries attention it needs to be able to take fully advantage of parallel computing hardware.

Expected Sarsa

The Expected Sarsa path tracer is no longer as trivial to parallelize as the default forward path tracing algorithm due to the the first addition to the algorithm detailed in section 3.1.2, where the incident radiance estimate for a given entry $Q(x, \omega_k)$ is updated after each intersection point is traced for constructing a light path. To parallelize the algorithm means to create and individual process to construct each light path as part of determining the estimate of a pixel value. However, the Q-table which stores the current approximation of all discrete incident radiance values in the scene is accessed by each thread. Meaning, it is possible that two threads will attempt to write to the same location at the same time in the Q-table if they intersect near one another, causing the update rule provided in Equation 2.27 will not be applied correctly. To remedy this, the update rule can be placed within a critical code section [49] to ensure only one thread at a time can calculate and set the updated incident radiance estimate. For my purposes, all code written for parallel execution used the CUDA Toolkit [42], where CUDA atomic operations were used to ensure only one thread at a time can calculate and apply the Q-value update. The critical section added a minor performance overhead to each thread in both waiting to and performing the update rule. However it was negligible when using a Q-table near the size of any used to render the images in 4.1, as due to the number of Q-values it is rare two threads attempt to write to the same address at the same time.

A significant performance penalty I found the Expected Sarsa method imposes on path tracing, is the time taken to perform the nearest neighbour search for finding the closest Irradiance Volume to a given intersection point in the scene. Whilst I implemented a KD-Tree on the GPU in order to reduce the nearest neighbour computational complexity to $O(\log n)$, I found the Expected Sarsa path tracer ran XX times slower then that of the default path tracer. The cause of this slowdown was found to be in per thread memory read times to global CUDA memory [44]. As a quick overview of CUDA's thread hierarchy; an execution thread in CUDA is contained within a group of threads known as a thread block. Each thread block is executed by a symmetric multiprocessor (SM) placed in a GRID. Following this, the memory hierarchy consists of per-thread local memory, per-block shared memory, and global memory which any thread can read and write to at any time. The issue with the Expected Sarsa path tracer is the Q-table is that the Q-table needs to be globally accessible to any thread in the GPU at any time, as every thread will be constructing a light path and during its construction it may update the value of any $Q(x, \omega)$ in the Q-table. This compounded with that the Q-table is far too large to store in per-thread or per-block memory for any commodity GPU, means nearest neighbour lookup performs $O(\log n)$ global memory lookups. Each uncached global memory lookup has roughly 100X higher latency then that of per-thread or per-block memory lookups [26]. Hence, the Expected Sarsa algorithm experiences a high performance penalty from idle time waiting for data to be returned from global memory. Note, NVIDIA claims that the Expected Sarsa path tracer is able to render images with the same number of samples per pixel only 20% then that of the default path tracer [16] to significantly reduce image noise, however they do not specify the details of the machines that were running these experiments. My implementation could be further optimised by ensuring aligned memory access and reducing the amount of data retrieved, however it is unlikely that any of these optimisations will be able to avoid the bottleneck of regularly querying global memory.

Neural-Q

As for the Neural-Q path tracer parallel execution is slightly different, whereby batches of light paths colour estimates are computed once at a time. Note, the batch size should be set to be large enough such that no processors are idle whilst computing a batch, whilst being able to hold the temporary approximated incident radiance values in memory which are used for importance sampling directions to continue light paths in. Also, mini-batches which are the same size as the batch of light paths can be used in querying and training the ANN by stochastic gradient descent. A problem which added a significant performance penalty on my approach compared to that in [39], is that there was no built in way to evaluate an ANN per-thread on the GPU using the Dynet framework, only code from the CPU could call any of the Dynet API functions. Meaning, my implementation had to frequently send data produced from tracing a batch of light paths to their next intersection point (positions, normals, throughputs etcetera) from the GPU to the CPU, only to call a dynet function which moved the data back to the GPU again for evaluating the ANN. Within Algorithm [**?**] there are a total of 3 calls to the Dynet API for evaluating the network for every batch of light ray paths, this means there is a total 6 batch data transfers from the host device memory (that directly accessible by the CPU) to GPU memory, as illustrated in **??**.

The latency added by this design flaw is mainly due to the peak bandwidth available for transferring data between memory in the GPU is 144 GB/s, while for transferring data from the CPU to the GPU (and vice versa) is only 8 GB/s [**?**]. Therefore, the 18x performance penalty received 6 times with each batch computation, makes evaluating my implementation of the Neural-Q path tracer on render time not a fair test. However, the performance penalty received by methods which use neural network based importance sampling for Monte Carlo path tracing have found that the cost of the evaluating the ANN without the data transfers from CPU to GPU (and vice verca) can still add a 10X peformance penalty compared to other importance sampling methods in some cases [39, 30]. Clearly this needs to be alleviated before the neural network importance sampling methods are adopted for Monte Carlo path tracing in industry. Answers to this problem may already lie in existing hardware, by leveraging the power of NVIDIAS TensorCores in Tesla GPUs [43], which claims to have 40X higher performance than CPUs for inference on ANNs.

4.4 Neural-Q Path Tracer Design

Move section down here discussing other methods

- With and without vertices in coord space for training and state reasoning
- Number of actions experimented with on door scene
- With and without epsilon greedy, lack of exploration

4.5 Recent Advancements in Neural Importance Sampling for Monte Carlo Path Tracing

Along with Nvidia's paper which suggested a reinforcement learning approach for approximating the incident radiance at any point in a scene [16], other reinforcement learning based methods for learning light transport in a scene have been engineered [38, 63], but none of them investigated the potential of deep reinforcement learning for this problem. However, recently neural networks for both the global and local sampling for light path construction have had a large amount of research [65, 39, 30, 27]. Global light path sampling is where the optimal density of sampled light rays per pixel is determined by the learning algorithm. Therefore, the reinforcement learning agent only determines where light paths should initially be sampled from. While this method does not reduce noise in the Monte Carlo path tracing to the extent local importance sampling does, both [38, 65] have applied this technique to reduce image noise with a much lower performance penalty compared to that of local importance sampling. I however, work on evaluating local light path sampling methods. Local light path sampling involves learning directions to continue light paths in when they intersect with surfaces in the scene, to efficiently guide them towards light sources.

The follow up paper to the one which introduced the Expected Sarsa path tracer [16] is [30]. Here, ANNs are experimented with for rendering in three different ways; for a given intersection point in the scene an ANN is trained to determine which light source should be used to compute the direct light incident on that intersection point, for approximating the visibility of an intersection point to all light sources, and for approximating the radiance at a given point in the scene to directly approximate a pixels colour value using many small ANNs. The results are promising for all three use cases, however unlike the Neural-Q path tracer, none of the methods estimate the incident radaince from a set of discrete directions around a point for importance sampling.

Another paper published by Disney's Zurich research team during the implementation of this thesis in Febuary 2019 introduced neural importance sampling [39], which samples a direction to continue a light path in and is known as Neural Path Guiding (NPG). This is currently the only other method which is an alternative for using neural networks to importance sample directions to continue light paths in. An ANN framework for modelling complex high-dimensional densities known as Non-linear independent Component Estimation (NICE) [18] is trained to learn the distribution of incident radiance at any given point in the scene. However, unlike my approach, the network takes in the intersection point $x \in \mathbb{R}^3$, intersection surface normal and the outgoing direction ω_o of interest to directly evaluate $L_o(x, \omega_o)$. A one blob encoding is then applied to these inputs which improves the speed and performance of inference by the network to output a single direction ω to continue the light path in. The probability density function over the hemisphere Ω of possible directions to sample from at an intersection point x can also be evaluated at ω using the proposed ANN framework. This has the advantage over my approach as it is able to learn the radiance distribution over the continuous set of directions in a hemisphere around the intersection point. Rather than discretizing the hemisphere into an adaptive quadrature to represent the radiance distribution as a discrete set of directions $omega_k \ \forall i = 0, ..., n$. However, the ANN framework proposed is far more expensive to evaluate than that used by the Neural-Q path tracer as it consisting of a one-blob encoding, many fully connected layers and a piecewise polynomial warp [39]. Whereas, the Neural-Q network in comparison only consists of two hidden layers, hence requires less memory for storing parameter values and evaluating the network compared to that in [39]. A comparison between the NPG and Neural-Q path tracers for their ability to reduce noise in Monte Carlo path tracing is an interesting area for future investigation.

Chapter 5

Conclusion

A compulsory chapter, of roughly 5 pages

The concluding chapter of a dissertation is often underutilised because it is too often left too close to the deadline: it is important to allocation enough attention. Ideally, the chapter will consist of three parts:

1. (Re)summarise the main contributions and achievements, in essence summing up the content.
2. Clearly state the current project status (e.g., "X is working, Y is not") and evaluate what has been achieved with respect to the initial aims and objectives (e.g., "I completed aim X outlined previously, the evidence for this is within Chapter Y"). There is no problem including aims which were not completed, but it is important to evaluate and/or justify why this is the case.
3. Outline any open problems or future plans. Rather than treat this only as an exercise in what you *could* have done given more time, try to focus on any unexplored options or interesting outcomes (e.g., "my experiment for X gave counter-intuitive results, this could be because Y and would form an interesting area for further study" or "users found feature Z of my software difficult to use, which is obvious in hindsight but not during at design stage; to resolve this, I could clearly apply the technique of Smith [7]").

5.0.1 Plan

1. Summarise contributions:
   (a) Implementing a path tracer from scratch to analyse in depth the difficulties and issues that come with Ken Dahm's algorithm. Including memory usage, parallelisation and parameter usage.
   (b) Analysis of different reinforcement learning approaches pitched together clearly on a variety of scenes
   (c) Analysis of neural networks ability to learn the irradiance distribution function
   (d) Online deep-reinforcement learning algorithms effectiveness of learning irradiance distribution function
2. If DQN does not work well provide some further analysis on potential other alternatives which could be used.
3. Future Work: Policy learning to model continuous action & state space
4. DDQN and other deep reinforcement learning strategies

Bibliography

[1] Opengl mathematics (glm).

[2] Simple directmedia layer. SDL version 2.0.9 (stable).

[3] TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[4] Erik Alerstam, Tomas Svensson, and Stefan Andersson-Engels. Parallel computing with graphics processing units for high-speed monte carlo simulation of photon migration. *Journal of biomedical optics*, 13(6):060504, 2008.

[5] Roger Allen. Accelerated ray tracing in one weekend in cuda. NVIDIA Developer Blog.

[6] Autodesk. Autodesk maya api white paper.

[7] 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. *ACM Trans. Graph.*, 36(4):97–1, 2017.

[8] Thomas Bashford-Rogers, Kurt Debattista, and Alan Chalmers. A significance cache for accelerating global illumination. In *Computer Graphics Forum*, volume 31, pages 1837–1851. Wiley Online Library, 2012.

[9] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[10] Bloomberg. Peak video game? top analyst sees industry slumping in 2019.

[11] 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. *ACM Transactions on Graphics (TOG)*, 36(4):98, 2017.

[12] Per Christensen, Julian Fong, Jonathan Shade, Wayne Wooten, Brenden Schubert, Andrew Kensler, Stephen Friedman, Charlie Kilpatrick, Cliff Ramshaw, Marc Bannister, et al. Renderman: An advanced path-tracing architecture for movie rendering. *ACM Transactions on Graphics (TOG)*, 37(3):30, 2018.

[13] Per H Christensen, Wojciech Jarosz, et al. The path to path-traced movies. *Foundations and Trends® in Computer Graphics and Vision*, 10(2):103–175, 2016.

[14] David Cline, Daniel Adams, and Parris Egbert. Table-driven adaptive importance sampling. In *Computer Graphics Forum*, volume 27, pages 1115–1123. Wiley Online Library, 2008.

[15] Thomas W Crockett. Parallel rendering. Technical report, INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERING HAMPTON VA, 1995.

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

[17] Luc Devroye. Nonuniform random variate generation. *Handbooks in operations research and management science*, 13:83–121, 2006.

[18] Laurent Dinh, David Krueger, and Yoshua Bengio. Nice: Non-linear independent components estimation. *arXiv preprint arXiv:1410.8516*, 2014.

[19] Philip Dutré, Henrik Wann Jensen, Jim Arvo, Kavita Bala, Philippe Bekaert, Steve Marschner, and Matt Pharr. State of the art in monte carlo global illumination. In *ACM SIGGRAPH 2004 Course Notes*, page 5. ACM, 2004.

[20] Kayvon Fatahalian, Edward Luong, Solomon Boulos, Kurt Akeley, William R Mark, and Pat Hanrahan. Data-parallel rasterization of micropolygons with defocus and motion blur. In *Proceedings of the Conference on High Performance Graphics 2009*, pages 59–68. ACM, 2009.

[21] Ian Foster. Designing and building parallel programs. Section 1.4.4.

[22] Iliyan Georgiev, Thiago Ize, Mike Farnsworth, Ramón Montoya-Vozmediano, Alan King, Brecht Van Lommel, Angel Jimenez, Oscar Anson, Shinji Ogaki, Eric Johnston, et al. Arnold: A brute-force production path tracer. *ACM Transactions on Graphics (TOG)*, 37(3):32, 2018.

[23] Andrew S Glassner. *Principles of digital image synthesis*. 2014.

[24] Gene Greger, Peter Shirley, Philip M Hubbard, and Donald P Greenberg. The irradiance volume. *IEEE Computer Graphics and Applications*, 18(2):32–43, 1998.

[25] Pat Hanrahan. Monte carlo path tracing. Stanford Graphics.

[26] Mark Harris. Using shared memory in cuda c/c++. NVIDIA Developer Blog.

[27] Pedro Hermosilla, Sebastian Maisch, Tobias Ritschel, and Timo Ropinski. Deep-learning the latent space of light transport. *arXiv preprint arXiv:1811.04756*, 2018.

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

[29] James T Kajiya. The rendering equation. In *ACM SIGGRAPH computer graphics*, volume 20, pages 143–150. ACM, 1986.

[30] Alexander Keller and Ken Dahm. Integral equations and machine learning. *Mathematics and Computers in Simulation*, 2019.

[31] Alexander Keller, Ken Dahm, and Nikolaus Binder. Path space filtering. In *Monte Carlo and Quasi-Monte Carlo Methods*, pages 423–436. Springer, 2016.

[32] George Konidaris, Sarah Osentoski, and Philip Thomas. Value function approximation in reinforcement learning using the fourier basis. In *Twenty-fifth AAAI conference on artificial intelligence*, 2011.

[33] Jaroslav Křivánek, Alexander Keller, Iliyan Georgiev, Anton S Kaplanyan, Marcos Fajardo, Mark Meyer, Jean-Daniel Nahmias, Ondrej Karlík, and Juan Canada. Recent advances in light transport simulation: some theory and a lot of practice. In *SIGGRAPH Courses*, pages 17–1, 2014.

[34] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[35] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[36] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[37] William J Morokoff and Russel E Caflisch. Quasi-monte carlo integration. *Journal of computational physics*, 122(2):218–230, 1995.

[38] Thomas Müller, Markus Gross, and Jan Novák. Practical path guiding for efficient light-transport simulation. In *Computer Graphics Forum*, volume 36, pages 91–100. Wiley Online Library, 2017.

[39] Thomas Müller, Brian McWilliams, Fabrice Rousselle, Markus Gross, and Jan Novák. Neural importance sampling. *arXiv preprint arXiv:1808.03856*, 2018.

[40] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[41] Graham Neubig, Chris Dyer, Yoav Goldberg, Austin Matthews, Waleed Ammar, Antonios Anastasopoulos, Miguel Ballesteros, David Chiang, Daniel Clothiaux, Trevor Cohn, Kevin Duh, Manaal Faruqui, Cynthia Gan, Dan Garrette, Yangfeng Ji, Lingpeng Kong, Adhiguna Kuncoro, Gaurav Kumar, Chaitanya Malaviya, Paul Michel, Yusuke Oda, Matthew Richardson, Naomi Saphra, Swabha Swayamdipta, and Pengcheng Yin. Dynet: The dynamic neural network toolkit. *arXiv preprint arXiv:1701.03980*, 2017.

[42] Nvidia. Cuda toolkit. Develop, Optimize and Deploy GPU-accelerated Apps.

[43] NVIDIA. Nvidia tensor cores. The Next Generation of Deep Learning.

[44] NVIDIA. Nvidia cuda c programming guide, 2012. NVIDIA CUDA.

[45] NVIDIA. *NVIDIA Turing Architecture Whitepaper*, 2018.

[46] Zhigeng Pan, Adrian David Cheok, Hongwei Yang, Jiejie Zhu, and Jiaoying Shi. Virtual reality and mixed reality for virtual learning environments. *Computers & graphics*, 30(1):20–28, 2006.

[47] Vincent Pegoraro, Carson Brownlee, Peter S Shirley, and Steven G Parker. Towards interactive global illumination effects via sequential monte carlo adaptation. In *2008 IEEE Symposium on Interactive Ray Tracing*, pages 107–114. IEEE, 2008.

[48] Ravi Ramamoorthi, John Anderson, Mark Meyer, and Derek Nowrouzezahrai. A theory of monte carlo visibility sampling. *ACM Transactions on Graphics (TOG)*, 31(5):121, 2012.

[49] Michel Raynal. *Concurrent programming: algorithms, principles, and foundations*. Springer Science & Business Media, 2012.

[50] Peiran Ren, Jiaping Wang, Minmin Gong, Stephen Lin, Xin Tong, and Baining Guo. Global illumination with radiance regression functions. *ACM Transactions on Graphics (TOG)*, 32(4):130, 2013.

[51] Barbara Robertson. Weighty matters, computer graphics world. Transformers: Revenge of the Fallen.

[52] Scratchapixel. Monte carlo methods in practice, Apr 2015.

[53] Peter Shirley and Kenneth Chiu. Notes on adaptive quadrature on the hemisphere. Technical report, Technical Report 411, Department of Computer Science, Indiana University , 1994.

[54] Walt Disney Animation Studios. Hyperion. Disney's Hyperion Renderer.

[55] Richard S Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Advances in neural information processing systems*, pages 1038–1044, 1996.

[56] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. Cambridge, MA: MIT Press, 2011.

[57] William TB Uther and Manuela M Veloso. Tree based discretization for continuous state space reinforcement learning. In *Aaai/iaai*, pages 769–774, 1998.

[58] Hado van Hasselt. Exploration and exploitation. DeepMind: Advanced Deep Learning & Reinforcement Learning.

[59] Hado van Hasselt. Function approximation and deep reinforcement learning. DeepMind: Advanced Deep Learning & Reinforcement Learning.

[60] Hado van Hasselt. Introduction to reinforcement learning. DeepMind: Advanced Deep Learning & Reinforcement Learning.

[61] Hado van Hasselt. Markov decision processes and dynamic programming. DeepMind: Advanced Deep Learning & Reinforcement Learning.

[62] Hado van Hasselt. Model-free prediction and control. DeepMind: Advanced Deep Learning & Reinforcement Learning.

[63] Jiří Vorba, Ondřej Karlík, Martin Šik, Tobias Ritschel, and Jaroslav Křivánek. On-line learning of parametric mixture models for light transport simulation. *ACM Transactions on Graphics (TOG)*, 33(4):101, 2014.

[64] Eric W Weisstein. Normal vector. MathWolrd–A Wolfram Web Resource.

[65] Quan Zheng and Matthias Zwicker. Learning to importance sample in primary sample space. *arXiv preprint arXiv:1808.07840*, 2018.

Appendix A

An Example Appendix

Content which is not central to, but may enhance the dissertation can be included in one or more appendices; examples include, but are not limited to

- lengthy mathematical proofs, numerical or graphical results which are summarised in the main body,
- sample or example calculations, and
- results of user studies or questionnaires.

Note that in line with most research conferences, the marking panel is not obliged to read such appendices.