

Surrounding-Aware Screen-Space-Global-Illumination using Generative Adversarial Network

by

Abrar Mahmud

18201147

Alimus Sifar

18201157

Moh. Absar Rahman

18201167

Fateen Yusuf Mostafa

18201200

Lamia Tasnova

18301053

A thesis submitted to the Department of Computer Science and Engineering
in partial fulfillment of the requirements for the degree of
B.Sc. in Computer Science

Department of Computer Science and Engineering
Brac University
September 2022

© 2022. Brac University
All rights reserved.

Declaration

It is hereby declared that

1. The thesis submitted is our own original work while completing degree at Brac University.
2. The thesis does not contain material previously published or written by a third party, except where this is appropriately cited through full and accurate referencing.
3. The thesis does not contain material which has been accepted, or submitted, for any other degree or diploma at a university or other institution.
4. We have acknowledged all main sources of help.

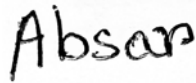
Student's Full Name & Signature:



Abrar Mahmud
18201147



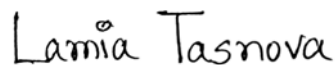
Alimus Sifar
18201157



Moh. Absar Rahman
18201167



Fateen Yusuf Mostafa
18201200



Lamia Tasnova
18301053

Approval

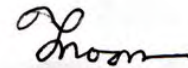
The thesis titled “Surrounding-Aware Screen-Space-Global-Illumination using Generative Adversarial Network” submitted by

1. Abrar Mahmud (18201147)
2. Alimus Sifar (18201157)
3. Moh. Absar Rahman (18201167)
4. Fateen Yusuf Mostafa (18201200)
5. Lamia Tasnova (18301053)

Of Summer, 2022 has been accepted as satisfactory in partial fulfillment of the requirement for the degree of B.Sc. in Computer Science on September 20, 2022.

Examining Committee:

Supervisor:
(Member)



Ms. Jannatun Noor Mukta
Senior Lecturer
Department of Computer Science and Engineering
Brac University

Program Coordinator:
(Member)

Dr. Md. Golam Rabiul Alam
Associate Professor
Department of Computer Science and Engineering
Brac University

Head of Department:
(Chair)

Dr. Sadia Hamid Kazi
Associate Professor and Chairperson
Department of Computer Science and Engineering
Brac University

Abstract

Global Illumination is a strategy in computer graphics to add certain degree of realism in case of 3D scene lighting, by trying to emulate how light rays work in real life. Several approaches exist to achieve such kind of visual effect for computed generated imagery. The most physically accurate approach is through ray-tracing. It can produce results which are realistic enough, with a trade-off of being time and computational-resource intensive, making them unsuitable for real-time usage. For more real-time usage scenarios, a set of faster algorithms exist that utilize rasterization rather than ray-tracing. Despite being faster, those still can be resource intensive or generate physically inaccurate results. Our Generative Adversarial Network based approach targets to bring close to physically accurate results based on rasterization output data which can be obtained from a conventional deferred rendering pipeline, while retaining speed. These rasterization output data, which are basically screen-space feature buffers will act as the input to our deep-learning network, which in turn will produce per-frame lightmaps that contain global illumination data, which are further used to generate a presentable frame on the screen. Using screen-space information from a single viewpoint won't always guarantee light consistency, thus our approach takes into account the rasterization output data of the surrounding of a certain viewpoint, producing more accurate global illumination.

Keywords: computer graphics, global illumination, gan, neural networks

Dedication

This dissertation is dedicated to our loved ones and close friends. A special feeling of gratitude to our devoted parents, whose words of encouragement and push for tenacity ring in our ears. We also thank Markus Alexej Persson and his team for creating the video game “Minecraft” which was an essential part of building our dataset. We will always be grateful for what they have done.

Acknowledgement

Firstly, all the praised to the great Allah for whom our thesis have been completed without any major interruption.

Secondly, to our supervisor Ms. Jannatun Noor Mukta for her kind support and advice in our work. She helped us whenever we needed help.

Finally, to our families and friends for all their support and prayers. Without their support, it would have been impossible to be here.

Table of Contents

Declaration	i
Approval	ii
Abstract	iii
Dedication	iv
Acknowledgment	v
Table of Contents	vi
List of Figures	viii
List of Tables	x
Nomenclature	xi
1 Introduction	1
1.1 Background and Motivation	2
1.2 Research Problems	3
1.3 Research Objectives	5
1.4 Our Contribution	5
1.5 Overview of Thesis	6
2 Background	8
2.1 Ray Tracing	8
2.1.1 Ray Tracing Texel eXtreme (RTX)	8
2.2 Rasterization	9
2.3 Global Illumination	9
2.4 Screen Space Global Illumination (SSGI)	9
2.5 Convolutional Neural Network (CNN)	9
2.6 Generative Adversarial Network (GAN)	10
2.7 PyTorch	11
2.8 Blender	12
2.9 Minecraft	12
2.10 Quality Evaluation Metrics	12
3 Related Works	14

4	Methodology	18
4.1	Design Principles	21
4.2	Dataset Generation	25
4.3	Dataset Pre-processing	28
4.3.1	Phase 1	28
4.3.2	Phase 2	31
4.4	Model Architecture	32
4.5	Training and Testing	37
5	Performance Evaluation	41
5.1	Experimental Configurations	41
5.1.1	Dataset Generation	41
5.1.2	Training Configuration	44
5.1.3	Testing Configuration	45
5.2	Experimental Results	45
5.2.1	Render Output Comparison	45
5.2.2	Visual Consistency Analysis	47
5.2.3	Quality Evaluation Metrics	50
5.2.4	Performance Evaluation Metrics	51
5.3	Experimental Findings	51
5.3.1	Findings About Quality Evaluation Metrics	51
5.3.2	Findings About Time Performance	52
5.3.3	Theoretical Comparison	52
6	Discussion	54
6.1	Limitations	54
7	Conclusion and Future Plans	56
7.1	Conclusion	56
7.2	Future Plans	56
	References	59

List of Figures

1.1	Scene lit with only direct light with on-screen light source (left). Scene lit with direct+indirect light with on-screen light source (center). Scene lit with direct+indirect light with off-screen light source (right).	3
1.2	Lighting inconsistency of screen-space global illumination (SSGI)	5
2.1	GAN Structure [34]	10
2.2	Languages used in PyTorch	11
4.1	General operation sequence of the system.	19
4.2	Work sequence of the research.	20
4.3	Comparison between direct and indirect illumination.[11].	21
4.4	Camera cluster	22
4.5	Demonstration of the Global View Merging Process	23
4.6	Coordinate Map, showing phi (ϕ) (in red) and theta (θ) (in green) coordinate angles.	24
4.7	Sample feature frame from the dataset.	27
4.8	Ground truth sample of a frame from the dataset	28
4.9	Sample frame from the 1st preprocessing stage output	30
4.10	Ground Truth Lumination	31
4.11	Sample frame from the 2nd preprocessing stage output	32
4.12	First image represents the generator model and second image represents the discriminator model	34
4.13	Generator loss.	40
4.14	Discriminator loss.	40
5.2	High-level compositor node setup for feature buffers	42
5.3	High-level compositor node setup for ground truth	43
5.4	Compositor nodes for feature buffer view routing	43
5.5	Compositor nodes saving feature buffer to file (shown for front camera)	44
5.6	The sunlight that hits the floor gets scattered multiple times within the room. (The hole in the wall is darker due to less light rays bouncing there.)	46
5.7	A scene mostly illuminated by sky scattering rather than direct sunlight.	46
5.8	The surfaces directly lit by sunlight are behind the camera, yet the surfaces visible by the main camera are luminated by indirect light.	46
5.9	Network generated result. Shows that the generated indirect lighting is dependent on diffuse surface color and direct color.	47

5.10	Moving camera, stationary light. For each pair, top is DeepIllumination [15] approach, bottom is Surrounding-Aware approach (ours) . . .	48
5.11	Stationary camera, moving light. For each pair, top is DeepIllumination [15] approach, bottom is Surrounding-Aware approach (ours) . . .	49
5.12	Structural-Similarity Index Measure.	50
5.13	Mean Squared Error.	50
5.14	Peak Signal-to-Noise ratio.	51
6.1	Incorrect Illumination Due to Light Source Obstruction	55

List of Tables

4.1	First 24 layers of Generator	35
4.2	Remaining layers of Generator	36
4.3	Parameter count of generator model	36
4.4	Layers of Discriminator	37
4.5	Parameter count of discriminator model	37
5.1	Hardware configuration used during training phase	45
5.2	Hardware configuration used during testing phase	45
5.3	Quality Evaluation Metrics comparison at 40th epoch.	51
5.4	Time Performance comparison	51
5.5	Comparison of Our Approach with Other Existing Approaches	52

Nomenclature

The next list describes several symbols & abbreviation that will be later used within the body of the document

API Application Programming Interface

DeepAO Deep Ambient Occlusion

GAN Generative Adversarial Networks

GPU Graphics Processing Unit

LPIPS Learned Perceptual Image Patch Similarity

MBTF Multi-scale Bidirectional Texture Functions

MSE Mean-Squared Error

NNAO Neural Network Ambient Occlusion

PSNR Peak Signal-To-Noise Ratio

RAD Rendering Aware Denormalization

RTX Ray Tracing Texel eXtreme

SPADE Spatially-Adaptive Normalization

SSIM Structural-Similarity Index Measure

Chapter 1

Introduction

The realism in the context of computer graphics can depend on several factors such as the properness of reflection and refraction, ambient occlusion, and global illumination. Real-time graphics are being used in various scopes other than just video games. For rendering vector graphics, rasterization has been the number one choice of application developers as it gives a reasonable performance, suitable for real-time usage. However, the rendered result can be relatively dull, unrealistic, and physically incorrect. Current rasterizing graphics APIs provide a highly-configurable rendering pipeline to develop and customize the rendering workflow. Such a pipeline can be utilized to implement the aforementioned graphics realism factors. For example, using cubemaps or screen-space data for reflection and refraction, ambient occlusion using screen-space data, and voxel-cone tracing for indirect lighting. Many of these approaches can bring a significant hit on performance when stacking one layer of processing on top of another, which can result in low frame rates, overheating of hardware, and high power usage. On top of that, using ordinary screen-space data can often generate spurious results, especially when an object of interest goes out of the screen or viewport.

On the other hand, a ray tracing-based renderer offers photorealistic results by following the laws of physical lighting, especially in the case of reflections, refractions, and global illumination. In ray tracing, the properties and transportation of photons are calculated as intended to travel in the real world. Even a basic form of ray tracer is much simpler to implement compared to its equivalent rasterizer. But there is a huge trade-off. The computational resource usage and time consumption of ray tracers are higher than a rasterized solution, making them unsuitable for the usage of real-time applications. Therefore, ray tracers are widely used where photorealism is given priority over rendering speed. Mostly architectural and artistic visualizations, film industry, etc.

However, certain amount of graphics processors are being developed in recent years, many of which are targeting the consumer markets, and are powerful enough to have the capability to perform ray tracing operations in real-time. They are usually used in such a way to assist the default rasterization-based pipeline to achieve certain effects, which a rasterizer by nature is incapable of, such as physically accurate real-time reflections and refractions. Another crucial lighting phenomenon, global illumination or indirect lighting, is achievable in real-time by these ray tracing ac-

celerated graphics processors. Still, one major drawback of this hardware is its price point. In addition to that, not every system is capable of cooperating with these devices. The scarcity and cost of these cards disable consumers of every level to experience real-time ray tracing. Surely, these devices are costly due to the amount and complexity of computations they need to perform to achieve their goal, which also adds up to power consumption.

1.1 Background and Motivation

The realism in the context of computer graphics can depend on several factors such as the properness of refraction and reflection, ambient occlusion, indirect lighting also known as global illumination. Real-time graphics are being used in various scopes other than just video games. For rendering vector graphics, rasterization has been the number one choice of application developers as it gives a reasonable performance that is suitable for real-time usage. However, the rendered result can be quite dull, unrealistic, and physically incorrect. Current rasterizing graphics APIs provide a highly-configurable rendering pipeline to develop and customize the rendering workflow. A such pipeline can be utilized to implement the aforementioned graphics realism factors. Examples, using cubemaps or screen-space data for reflection and refraction, ambient occlusion using screen-space data, and voxel-cone tracing for indirect lighting. Many of these approaches can bring significant hits on performance when stacking one layer of processing on top of another, which can result in low frame rates, overheating of hardware, and high power usage. On top of that, using ordinary screen-space data can often generate spurious results, especially when an object of interest goes out of the screen or viewport.

On the other hand, a ray tracing-based renderer offers photorealistic results by following the laws of physical lighting, especially in the case of reflections, refractions, and global illumination. In ray tracing, the properties and transportation of photons are calculated as they are intended to travel in the real world. Even a basic form of ray tracer is much simpler to implement compared to its equivalent rasterizer. But there is a huge trade-off. The computational resource usage and time consumption of ray tracers are extremely high. These are a lot higher than the rasterized solution. That is why it makes them unsuitable for the usage of real-time applications. Ray tracers are widely used where photorealism is given priority over rendering speed. Mostly architectural and artistic visualizations, film industry, etc. Figure [1.1] shows global illumination, an aspect covered by ray tracing. Here, only applying direct light can show unrealistic result, but incorporating indirect light as well brings more realism. In ray tracing, it does not matter if lights are visible on the screen, a physically correct lighting simulation will be calculated whatsoever.

However, a certain number of graphics processors are being developed in recent years, many of which are targeting the consumer markets, and are powerful enough to have the capability to perform ray tracing operations in real-time. They are usually used in such a way to assist the default rasterization-based pipeline in order to achieve a certain amount of effects, which a rasterizer by nature is incapable of, such as physically accurate real-time reflections and refractions. Another crucial lighting phenomenon that is global illumination or indirect lighting is achievable



Figure 1.1: Scene lit with only direct light with on-screen light source (left). Scene lit with direct+indirect light with on-screen light source (center). Scene lit with direct+indirect light with off-screen light source (right).

in real-time by these ray tracing accelerated graphics processors. Still, one major drawback of this hardware is its price point. In addition to that, not every system is capable of cooperating with these devices. The scarcity and cost of these cards disable consumers of every level to experience real-time ray tracing. Lastly, these devices are costly due to the amount and complexity of computations they need to perform to achieve their goal, which also adds up to power consumption.

1.2 Research Problems

Ray tracing is an eye-focused procedure that involves traveling through each pixel in search of the object that should be displayed there. It is recognized as the best method for image creation and is well-known in the film industry because it represents the reflection of light from all surfaces, creating the graphical style that everyone is familiar with. As a result, ray tracing can improve the quality of each frame. However, it is more challenging to perform in real-time than rasterization because while it is easy to track and process one ray, it becomes a problem when the ray bounces off of a surface as it turns into a countless number of reflected rays. This rise is exponential, and calculating all of these rays will take a long time and require more GPU power leading to more expense.

The computer capacity required to carry out real-time ray tracing is expensive at a cost that would allow widespread adoption, which is why it took so long to enter the game industry. The base price is prohibitively high—neither AMD nor Nvidia provides a low-cost graphics card that supports hardware accelerated real-time ray tracing. It is important to remember that real-time ray tracing is still a relatively new technology in the video game business because displaying an entire game in real-time ray tracing is still well beyond today’s hardware capabilities. Ray tracing is only used in games that support it for a few effects, mostly shadows, lighting, some parts of reflection, and refraction while everything else is still rasterized. There will always be a performance trade-off for the current console generation.

The authors of this paper [1] have observed some drawbacks of ray tracing are as follows: wastage of time in calculating intersections between rays and objects, ray tracing algorithm not being able to take advantage of coherency, rendering time not improving one bit even after using object coherency and hierarchical clustering objects, and for anti-aliasing, adequate information is not correlated with each ray. For rendering simple scenarios, 75% of the total time is used up to calculate the intersections between rays and objects. For complex scenes, the time spent is 95%. Other rendering methods use the advantage of coherency. However, with ray tracing, it is much more difficult to do it. When a ray is being traced, it happens independently, without using the benefits of the data created by the neighboring rays.

Earlier development happened based on one of the two methods, either object coherency or hierarchical clustering of objects. Despite using either of the methods, ray tracing seems to have no improvements in rendering time. The challenging issue with anti-aliasing in ray tracing is that it does not provide enough details with an individual ray. Rays allow us to sample a single spot in the pixel's center. There is no way for us to know or calculate what else is visible in the vicinity surrounding the sample location without emitting additional rays.

Opposed to ray tracing, the voxel-cone tracing approach can be used to produce indirect illumination for rasterized graphics which was demonstrated in this paper [5]. It incorporates a pre-filtered hierarchical voxel octree representation of a 3D scene and bounces off two types of light rays through it, one for Lambertian surfaces and another one for specular ones. The indirect lighting is produced by calculating the path of these bounced lights within the voxel octree, resulting in interactive global illumination with temporal coherence and free from noises. However, it comes with some costs. For instance, if there are objects within the scene that are moving from frame to frame, the voxel octree needs to be recalculated, which can incur a certain degree of performance penalty. Furthermore, this approach suffers from leakage of light.

Reusing data or pixel features from already rendered frames in real-time 3D applications is a cheap way to add a few aspects of realism to the rendered frames, known as the screen-space approach. It is majorly used in the gaming industry and game engines. The frames which are presented on the screen are composed of multiple sequential render passes. The images produced in a render pass can be used to add certain effects in the subsequent render passes. This technique falls into the post-processing category and is used to add visual effects like reflections using Screen-Space Reflections (SSR), Screen-Space Ambient Occlusion (SSAO), and Screen-Space Global Illumination (SSGI). While it is a low-cost solution to many problems, the disadvantage of this method is that it can generate effects from only the objects visible on the screen. Any object that goes out of the screen or viewport will be unable to cast effect, creating temporal inconsistencies. Figure 1.2 contains some frames from a video by Default Cube on YouTube[27] that shows the major drawback of screen-space global illumination. Notice the interior of the room only gets lit enough when the light source (here the window) is visible on the screen, but completely dark when it is off-camera.

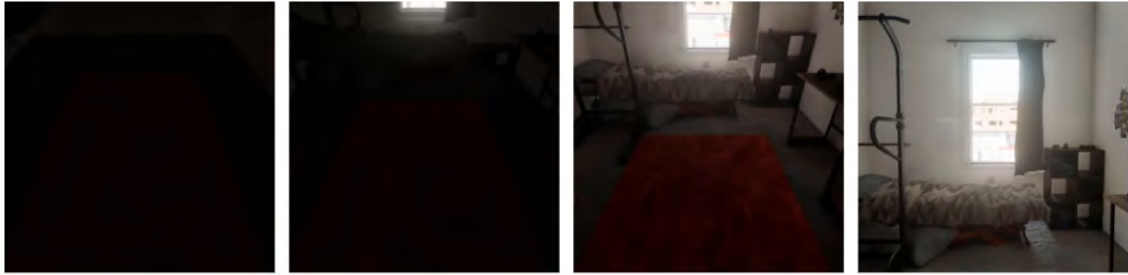


Figure 1.2: Lighting inconsistency of screen-space global illumination (SSGI) [27]

Another naïve approach to realism, which is mostly used for global illumination purposes, is using baked textures or lightmaps for scene rendering. While it incurs almost no additional performance cost, it fails to render scenes with dynamic lights and/or objects properly, making its use suitable only for static scenes.

1.3 Research Objectives

Our research aims to work on developing a post processing system for rasterized 3D graphics in order to enhance the visuals of rendered scenes by applying global illumination on the frames, through analyzing the surrounding per-pixel geometric features and lighting information with the help of an image generating neural network. Conventional deferred renderers are able to generate these rendered features with ease. These features can be used as input for the visual enhancement system.

Our research objective follows:

1. Understanding image generating and image transforming neural networks, especially GANs.
2. Determining the best fit approach for visual enhancing via diffuse global illumination.
3. Generate dataset suitable for model training.
4. Develop the model and perform training.
5. Test the model against novel sample data.
6. Determine the drawbacks of the model and suggest possible ways for improvement.

1.4 Our Contribution

As we have mentioned earlier in Research Problems regarding multiple drawbacks. Firstly, conventional ray tracing is resource intensive for real-time usage. Secondly,

hardware ray tracing supported GPUs are expensive. Last but not least conventional Screen-Space Global Illumination suffers from incorrect lighting in case of off-screen light sources. Out of all the drawbacks, we are focused on solving the last one in our current research. Based on our target, we have put together our contributions:

- We have designed our proposed model in such a way that it tries to prevent incorrect lighting scenarios in the case of off-screen light sources when it comes to conventional implementation of screen space global illumination.
- Conventional raytracing process for global illumination is resource-intensive and time-consuming to operate. We try to propose an approach for post-processing global illumination systems for rasterization pipelines.
- Screen-space based global illumination solutions suffer from inconsistencies due to objects going out of the viewport. We redefine the approach by utilizing off-screen information.
- We use a deep learning approach that is fast enough and trained on ray-traced renders, delivering realistic post-processing effects while being suitable for real-time applications.
- We compare our approach with existing screen-space global illumination techniques to observe the output quality of each.
- We test our solution on various consumer-grade hardware to analyze the latencies and suitability for real-time usage.

1.5 Overview of Thesis

In chapter 2, we will discuss about our background. The chapter will demonstrate the technologies, concepts, and libraries used in our research such as Ray tracing, Global Illumination, GAN, CNN, etc. After that, in chapter 3 we will discuss related works. Here, we will present some rundown discussions about existing literature corresponding to our work. Initially, we will look at global illumination and how this technology has brought a revolutionary change in computer graphics. After that, we will be looking into more novel approaches that brought a remarkable leap not only in computer graphics but also in the world of computer science. These state-of-the-art technologies are one of the reasons why many developers and researchers can contribute and build more high-end technologies beyond human expectations. In chapter 4, we will be scrutinizing our objective and our approaches to generating our datasets and designing our system to solve the problem on which we are focused. We will also discuss how our data is preprocessed. After that, we will be demonstrating how we divided our datasets and the ratio of training and testing from the dataset. Moreover, we will be discussing how we build our proposed system and use our dataset to train and validate. Finally, we will be demonstrating the theoretical comparison. In chapter 5, we will analyze and compare our results using our proposed system with an existing deep learning-based screen space global illumination system. We will be using SSIM, MSE, and PSNR for our objective evaluation. Later, we will do a performance evaluation between our proposed system, the existing system, and Ray traced (Blender) using Nvidia Geforce GTX 1660

Ti and Nvidia Geforce GTX 1050 Ti. In chapter 6, we will give an overview of our paper and the limitations of our proposed model. Finally, in chapter 7, we will be discussing our thoughts, limitations, and plans for our future research.

Chapter 2

Background

In this chapter, we will discuss technologies and modules that are commonly used in our research such as ray tracing, rasterization, global illumination, screen space global illumination, convolutional neural network, generative adversarial network, and last but not least, PyTorch.

2.1 Ray Tracing

Ray tracing [4] is a widely used technique for creating high-fidelity and realistic computer graphics. Moreover, it is used to solve complex problems and equations in modern times. Ray tracing is costly even though it is a highly parallelizable algorithm. Modern games are using ray tracing to use the full potential and flexibility offered by the GPU.

2.1.1 Ray Tracing Texel eXtreme (RTX)

In the modern era, the demand for graphical tasks is increasing exponentially. Researchers are finding ways to improve how to utilize graphics to compute complex problems and find an optimal solution. According to this paper [24], the competition and the demand for GPUs are extremely high which resulted in a record revenue of \$11.72 billion which was achieved for Fiscal 2019 compared to Fiscal 2018, which was about \$9.71 billion. Furthermore, engineers are also aiming to increase the capabilities of GPUs. The RTX series is a good example of GPU development. Researchers, developers, and other professionals are using RTX to solve complex problems in a short amount of time. The researchers are still attempting to reduce the rendering latency.

RTX platform combines ray tracing, deep learning, and rasterization to structurally transform the creative process for the developers. This platform offers API and SDKs which allows the users to run it on advanced hardware for yielding solutions that are capable of accelerating and strengthening images, graphics, and video processing.

2.2 Rasterization

It [14] is a technique by which it takes a vector primitives and converts it into a set of pixels to the output. It also points to the technique where 3D models are converted into 2D projections.

2.3 Global Illumination

Global Illumination [3] is a technique by which it models how it should bounce off light from one surface to another instead of capping itself to direct light. Global Illumination has two steps:

1. Direct Light
2. Indirect Light

Direct Light

The light which comes directly from the light source is known as direct light. When light heads on towards an object, referred to as direct light. It is easy to calculate compared to indirect light.

Indirect Light

Indirect light is the reflected light that comes out from the surface. In other words, when the light hits an object and it bounces off of the surface of that object then that light is known as indirect light. Indirect lights are more complex than direct lighting since it needs to handle multiple cases, especially how the light should bounce off and which coordinates it should bounce off. In many applications, the developers decide to handle indirect light in many ways. Light maps are commonly used for handling indirect light.

2.4 Screen Space Global Illumination (SSGI)

SSGI [35] [36] is a feature commonly used by game engines for creating natural-looking lighting by including dynamic indirect lighting to objects by using the depths and color buffer of the viewport to calculate the diffuse light bounces.

2.5 Convolutional Neural Network (CNN)

CNN [9] is a neural network that consists of one or multiple convolutional layers. The layers of CNN consist of an input layer, hidden layers, and an output layer. It is frequently used in classification, image processing, NLP, and other applications. Each convolutional layer consists of filter sequences addressed as convolutional kernels. The filter consists of integer matrices utilized on the input's subset pixel value which is the same size as the kernel. Later, the result is aggregated into a single

value for the sake of simplicity to represent a pixel in the output map. This output consists of a series of feature maps. A feature map is an output generated by applying one filter to the previous layer. It is used to identify divergent features which are available in the image. The goal of the pooling layer is to continuously deduct the structural size of the image given as an input to reduce the computational numbers in the neural network. These layers are added in between convolutional layers. Max pooling is a commonly used pooling approach.

2.6 Generative Adversarial Network (GAN)

GAN [7] is an unsupervised deep learning model which can automatically discover and find patterns from the input data. After finding out those patterns, the model can be utilized to generate new output. It was initially proposed by Ian et al [7]. GAN model has two sub-models:

1. Generator
2. Discriminator

In Figure 2.1, the authors [34] demonstrated how the whole structure of GAN.

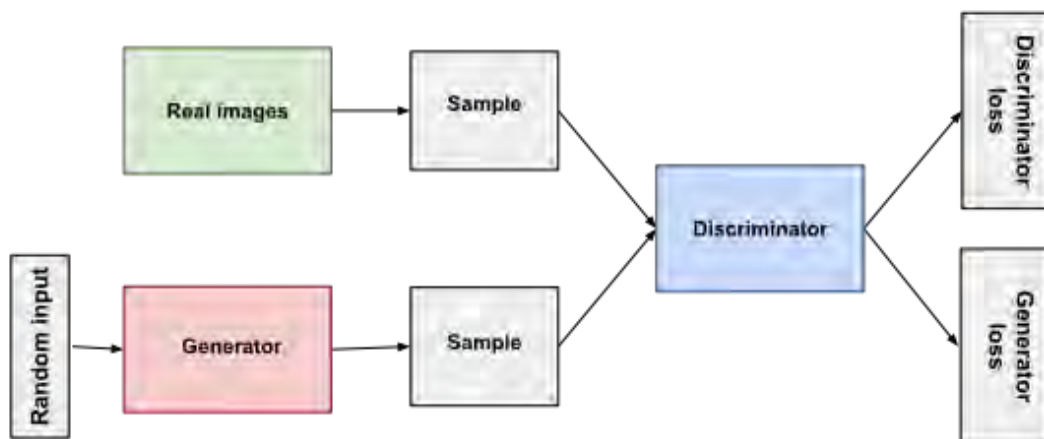


Figure 2.1: GAN Structure [34]

Generator

The verbal meaning of a generator is to generate something. In the case of the GAN model, the generator is responsible for generating synthetic data based on the input feature image buffer fed to the model which we can see in Figure 2.1.

Discriminator

A discriminator is responsible for recognizing the real data from the output data generated by the generator. In other words, it classifies both real and synthetic

data. After that, it updates and assigns new weights by utilizing back-propagation via the discriminator loss through the discriminator network visualized in Figure 2.1.

2.7 PyTorch

An open-source framework [21] widely used in machine learning. The framework is based on the Torch library and is commonly utilized in computer vision, and NLP. This framework is responsible for optimizing complex calculations and linear algebra by exploiting with or without CUDA. Currently, more than 159k users use this framework. It is contributed by more than 2.4K. One of the reasons why PyTorch is preferable is because of the massive community. The documentation is readable and well organized. As a result, it is easier for the users to use PyTorch. Moreover, it is flexible to use. PyTorch also supports data parallelism meaning its execution is asynchronous.

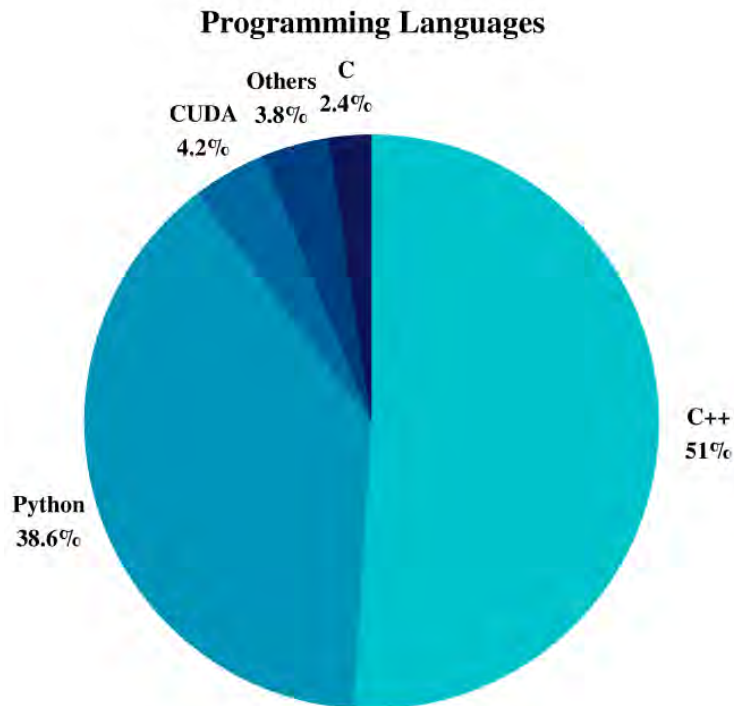


Figure 2.2: Languages used in PyTorch

2.8 Blender

Blender [16] is a cross-platform open-source 3D modeling software that offers various features beyond modeling like rigging, sculpting, animation making, compositing, video editing, etc. Along with a highly customizable user interface, it comes loaded with a built-in Python interpreter, which enables us to extend its functionality even more. Blender is capable of both rasterized and ray traced renders, using the built-in Eevee and Cycles render engines, respectively. Using its compositing feature, one can view, manipulate, combine, and save intermediary graphical feature buffers produced and used during the rendering process.

2.9 Minecraft

Minecraft [33] is an open-world video game of the sandbox variety developed by Mojang Studios. It enables a player to roam around within a randomly generated world. In that world, the player has to survive and create things. It provides an achievement system and a variety of resources. The player has the freedom to manipulate objects however he/she wants. Among the two ports of the same game, Java port enables the player to modify the game through countless mods. Even though it is mainly used as a game, it has been subject to scientific research [29].

2.10 Quality Evaluation Metrics

The common metrics used for analyzing outputs generated by global illumination approximating algorithms.

Mean Squared Error (MSE)

MSE [31] is one of the most widely used metrics to measure the performance of the metrics. The measurement is done by getting the average squared value of the difference between the model's prediction and the ground truth. The equation 2.1 demonstrates how MSE is calculated.

$$MSE = \frac{1}{n} \sum_{i=1}^n (Y_i - \hat{Y}_i)^2 \quad (2.1)$$

Where n is the number of data points, Y denotes the vector of observed values for the predicted variable, and \hat{Y} denotes the predicted values.

Structural Similarity Index Measure(SSIM)

SSIM [2] is a commonly used metrics. It is used to measure the intuitive difference between two homogeneous images. In this equation 2.10, SSIM takes two parameters for continuing the calculation.

$$SSIM(x, y) = \frac{(2\mu_x\mu_y+c_1)(2\sigma_{xy}+c_2)}{(\mu_x^2+\mu_y^2+c_1)(\sigma_x^2+\sigma_y^2+c_2)} \quad (2.2)$$

Where μ_x and μ_y are the pixel sample means of x and y , σ_x and σ_y are the variances of x and y , σ_{xy} is the covariance of x and y respectively and finally, c_1 and c_2 are two constants that maintains the stability once the denominator tends to be 0 (zero).

Peak Signal-to-Noise Rate (PSNR)

PSNR [28] is widely used for measuring the restoration quality for lossy image compression. During image compression, many noises are introduced. In other words, it figures out the ratio between the maximum possible potential of the original image and the corrupting noise power which infects the quality of the image. Equation 2.10 is used to calculate PSNR.

$$PSNR = 20 \cdot \log_{10}(MAX_I) - 10 \cdot \log_{10}(MSE) \quad (2.3)$$

Where MAX_I and MSE are the maximum possible pixel value of the image and the *Mean Squared Error* respectively.

Chapter 3

Related Works

This section examined previous relevant work on the subject of image composition using deep neural networks. Since our objective is to develop a post-processing system that produces global illumination lightmaps with the help of deep learning networks, it is necessary to correlate and compare our goal with existing deep learning-based image enhancement techniques. Several of these, however, do not tend to generate global illumination specifically, which is our target but follow their own means of image enhancement. Yet, what these approaches have in common is that they take one or multiple images as input, transform them, and produce an enhanced output image based on the input data. Besides these, we looked into a few non-deep-learning and non-machine-learning-based approaches as well, as soon as those remain relevant to our goal.

Rather than rendering every aspect of the final output of a scene using neural networks, certain single features of lighting, such as only ambient occlusion can be evaluated and post-processed onto the intermediary rasterized output. According to the authors in this paper [12], camera space normals and scene depth could be utilized to train a neural network against physically accurate ray traced renders of global ambient occlusion. The final trained network could be evaluated during runtime which would be equivalent to a single shader-pass. The rapidness of this neural network-based ambient occlusion (NNAO) calculation enables it to compete with existing screen-space ambient occlusion techniques.

According to this paper [26], the authors demonstrated how Deep Ambient Occlusion (DeepAO) reflects an improvement to the previously mentioned Neural Network Ambient Occlusion (NNAO). While NNAO runs fast due to its limited dataset, it cannot deliver desirable results for scenes with diverse objects. This approach makes the use of a convolutional neural network implementation using compute shaders runnable in the GPU to perform pooling, activation, batch-normalization, up-sampling, and concatenation. The encoder and decoder parts are packaged as a shader library. This network, like other methods, also utilizes G-Buffers. By training the it with a large dataset with a diversely classified object, DeepAO delivers better results compared to the prior approach.

Rather than using generative adversarial networks for resolving screen space global illumination, the paper [22] follows a different approach by training a very simple

multi-layered convolutional neural network which consists of a feature extractor and an image reconstructor. The network is trained with feature buffers and ground truth which are the corresponding path traced images. This rather simple network calculates indirect lighting pretty close to ground truth. However, it cannot resolve light bleeding from off-screen objects.

Contrary to ambient occlusion, another way of graphical realism is rendering the scene with proper indirect lighting. Conventional methods for calculating indirect or global illumination in real-time using the GPU like voxel cone tracing can be fast but with the cost of precision, compared to path tracing. The work, Deep-Illumination [15] proposes a method of evincing indirect lighting along with soft shadows by taking advantage of deep learning. Where the network is a U-Net, trained with pre-computed path traced and voxel-cone tracing global illumination, renders mapped against G-Buffers, which are frames containing screen-space diffuse color, depth information, normals, and direct lighting information. In this experiment, the authors have done four experiments based on their previous experience. 1) Reducing mean squared error and perceptual error by increasing the number of iterations while training, 2) Producing a better approximation by including a large amount of light object camera configuration, 3) Resulting faster convergence while training by increasing the layer numbers that will cost execution time, and 4) creating a neural network-based on different ground truths. Upon training, the network can approximate global illumination for dynamically and randomly placed lights, cameras, and objects; configurations foreign to that of the training phase.

The authors in this paper [25] suggested a way to compute indirect illumination using convolutional neural networks based on a U-Net architecture. The network is modified to have bilateral convolutional layers instead of ordinary ones. The approach takes advantage of a conventional OpenGL renderer to prepare direct lighting maps and G-Buffers. Later on, these are passed on to the neural network to determine a low-resolution indirect illumination map, which is subsequently upscaled by albedo modulation and joint bilateral technique. The process is performed fast enough to be implemented in interactive applications.

In the following paper [23], the authors used a conventional deferred rendering pipeline to make use of the screen-space feature buffers which were produced during runtime. The different phases of the rendering pipeline can be employed to train a higher-dimensional neural network-based texture, which can be evaluated to re-construct photorealistic graphics. This special neural texture can convey more appearance and geometric feature information as opposed to a basic low-dimensional feature map. In a similar fashion to how the basic feature maps are interpreted by shader programs, the new neural alternative can be similarly evaluated in the graphics pipeline. The underlying 3D scene may be less-detailed, but the output acquired from the learned network would have more realism.

Spatially-Adaptive Normalization, often known as SPADE, is a conditional normalization technique used in semantic image synthesis. Identical to batch normalization, channel-wise normalization of the activation is followed by modulation with learned scale and bias. To create the modulation parameters γ and β in the SPADE, the

mask is initially projected onto an embedding space and then convolved. In contrast to earlier conditional normalization techniques, γ and β are tensors with spatial dimensions rather than vectors. The generated γ and β are multiplied and added elementally to the normalized activation.

Following a different approach from pix2pixHD [18], the authors [20] proposed a rather novel technique for developing photorealistic images from semantically labeled maps, where each color represents a different type of object in the scene. While pix2pixHD [18] relies on a conditional normalization layer, which has a tendency to wash away semantic information if used with segmentation maps, in Spatially Adaptive Modulation the semantic information does not decay like they do in prior approaches. In the generator of the GAN, there is no need for an encoder as the SPADE already bears enough encoded information, which makes the whole network a lot lighter. Here, in the generator, a series of residual blocks consisting of the aforementioned spatially adaptive modulators are arranged with upscaling layers in-between. This arrangement gradually upscales the features while preserving consistency with the labeled map. These residual blocks are fed with the semantically labeled map each with their respective resolution. The final layer outputs the generated photorealistic image.

The paper [29] shows work on training realistic image generation techniques based on unsupervised training. Meaning, the training data set is unpaired with the 3D projected view. Rather than relying on absolute ground-truth data, pseudo ground-truth training images generated using the SPADE image synthesis model by giving semantically labeled projected views of the 3D viewspace are employed for training. The final rendered images are view-independent and frame-consistent. Meaning the output scene frames will be congruous regardless where the scene is being viewed from.

The author of the paper [32] shows another method of training an image enhancement network to achieve photorealistic results using unpaired images. Even with the absence of conducive reference images, the network learns to record the unique styles of dataset images. And the learning of suitable correspondences happens implicitly. The G-Buffers containing geometric, lighting and material information of 3D scenes produced by a conventional rendering pipeline are utilized here. A G-Buffer encoder network analyzes these buffers to produce multi-scaled features. Which are then passed on to an image enhancement network, which generates a comparatively photorealistic image. The image is further judged by an image patch perceptual similarity metrics system called LPIPS, which measures how much consistency there is between the input and output image. Another judgment is done by a trained perceptual discriminator, which measures the realism of the output image. These two metrics scores are then used to penalize the enhancement network. In the enhancement network, image features are modulated by Rendering Aware Denormalization (RAD), which convey better modulation results compared to the previously mentioned SPADE.

The paper [19] stated about how the authors used transferring GANs to generate images by exploiting limited amounts of data. In the sense of generating fine de-

tailed imagery from rather simplistic and less-detailed visual patterns, Generative Adversarial Networks (GANs) can be utilized. They consist of a generator, responsible for trying to develop an image for a particularly given input scenario and a discriminator, which critiques the generator on the generated image being real or fake. However, basic GANs are confined to small and simple datasets, and resolving them may involve the use of convolutional architectures, in which the image is up-sampled in the generator. Deep Convolutional Generative Adversarial Networks (DC-GAN) follows exactly that path for image construction.

The authors [30] proposed NeuMIP for representing and rendering material appearances variation at independent scales. They generalized conventional neural textures mipmap pyramids to pyramids which are combined with fully connected networks. The authors showed that their neural architecture studies Multi-scale bidirectional texture function (MBTF) with two dimensions for each query location, both incoming and outgoing direction and one dimension for the refined kernel radius. They have achieved real-time performance in their implementation of OptiX.

Chapter 4

Methodology

The objective of surrounding aware screen-space global illumination is to produce comparatively realistic lightmaps from the multi-view screen-space geometric, material, and lighting maps (G-Buffers or Feature-Buffers), as delineated previously. The system will utilize 360° images as inputs to generate global illumination lightmaps, which attempt to overcome the limitations of existing screen-space global illumination solutions. In order to perform the above-mentioned process, the system needs to acquire the Feature-Buffers, generated by conventional deferred rendering pipelines on a per-frame basis, using simple and common shader programs. To train the neural network, the ground-truth lightmap image, which contains both direct and indirect lighting information of a particular 3D scene frame, will be collected, and a ray tracer will render rather than a rasterizer to preserve realism. The generated image is considered ground truth, against which the neural network is to be trained and tested. The sequence below shows a workflow of training and testing the network.

In brief, the steps for designing the neural post-processing network follows:

1. Dataset Preparation: We have created 3D sample scenes in software like Blender and rendered the scenes from different viewpoints with varying environmental conditions. The feature buffers (depth, normals, albedo/flat-color, and emission maps) will accompany the renders. All images will be in the same dimension.
2. Pre-processing: The pixel values of the images will be transformed into a format that is easier for neural networks to ingest. This is our generated hard-paired dataset.
3. Dividing: We create the dataset in two parts, one for training and the other for testing.
4. Building the model and training: We plan to build our neural network following GAN architecture and train it using our generated training dataset.
5. Testing: The testing will be done mathematically based on how much enhanced the generated images feel to the human eye and how consistent the outputs are with the inputs.

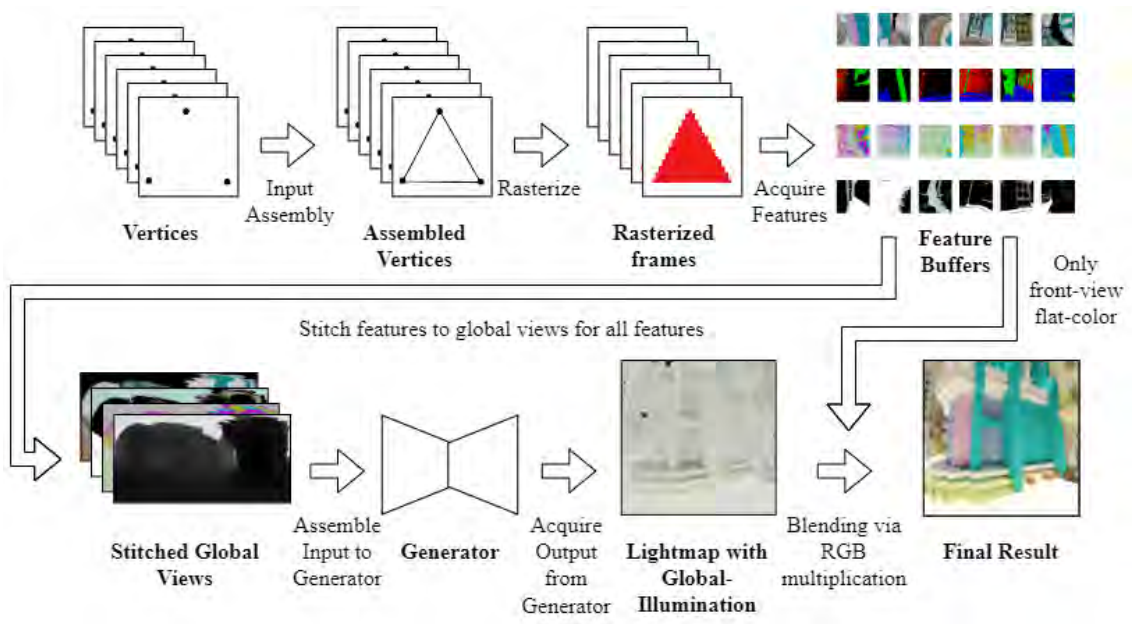


Figure 4.1: General operation sequence of the system.

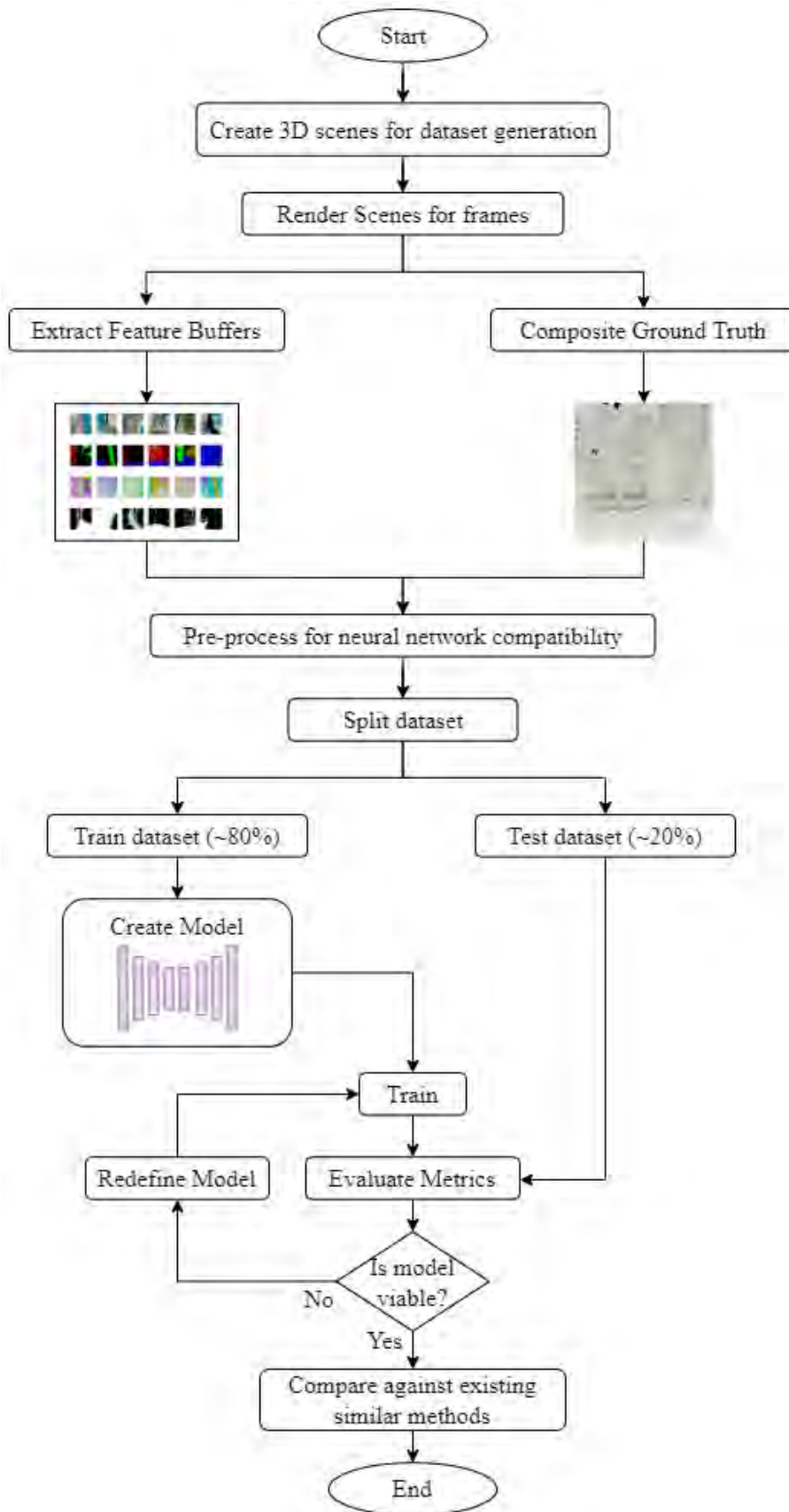


Figure 4.2: Work sequence of the research.

4.1 Design Principles

The approach we follow to achieve realism in rasterized renders through post-processing is by implementing realistic lighting. Generally, there are two types of lighting in 3D scenes. Direct lighting means light rays from an explicit light source, such as a point-light, a spotlight, or a directional light, hit a surface, illuminate it, and ultimately reach the viewer (camera or eye). This form of lighting is easy and fast to calculate by ordinary hardware and renderers. The other type is indirect lighting, which is light bounced from lit surfaces illuminating other surfaces in the scene, where the bounce can happen several times. This form is tough to calculate in real-time through ray tracing by conventional graphics hardware, and existing screen-space global illumination techniques suffer from light loss as explicit light sources go out of view. Direct illumination alone cannot bring proper realism to renders. A visualized comparison between direct illumination and indirect illumination in Figure 4.3 [11].

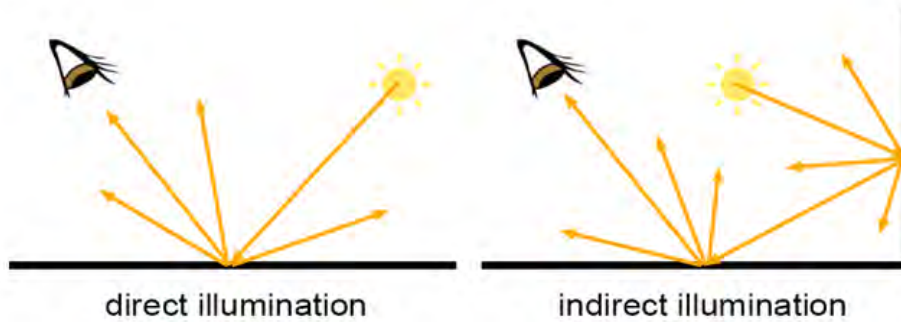


Figure 4.3: Comparison between direct and indirect illumination.[11].

A neural network solution for this can be a fast approach to this problem, given that all graphical features for a particular rendered frame from the viewpoint of a camera, also known as screen-space data, is readily available. These features include: direct lightmap is the image containing the directly lit surfaces by light sources; diffuse color is the image of the flat color of each surface in the scene; emission map is the image of surfaces that emit light on their own; environment map is the image of the light scattered from the sky; normal map is the per-pixel normal vector directions of each surface; and depth map is the per-pixel distance of each surface from the viewing point. The neural network can analyze the directly-lit surfaces from the features and predict the indirect lighting. If a directly-lit surface or emissive object is not within the camera’s point of view (for example, only the objects behind the camera are lit, but not visible to it), the network will not be able to predict the desired indirect lighting, due to a lack of information. This is a drawback of the above-mentioned approach. The fix is to give the network enough information, in this context, to provide the visuals of the surroundings and not just what is in front of the camera, making the system “surrounding-aware”. Our approach does exactly that by capturing 360° imagery for each frame and utilizing a collection of cameras rather than a single camera. This collection is called a “camera-cluster”, a rigid arrangement of six cameras looking in six directions (front, back, left, right, up,

down), adjacent ones placed 90° from each other, in a cube-like fashion. For this experiment, we set each view to have an aspect ratio of 1:1 and each camera to have a 90° field-of-view (FOV). For moderately high-resolution rendering, it would be sufficient to employ input feature buffers of the fractional size of the original screen presentation resolution since we are generating global illumination lightmaps, where sharpness is not a priority. If needed, the resulting low-resolution global illumination lightmaps can be up-scaled through 2D interpolation.

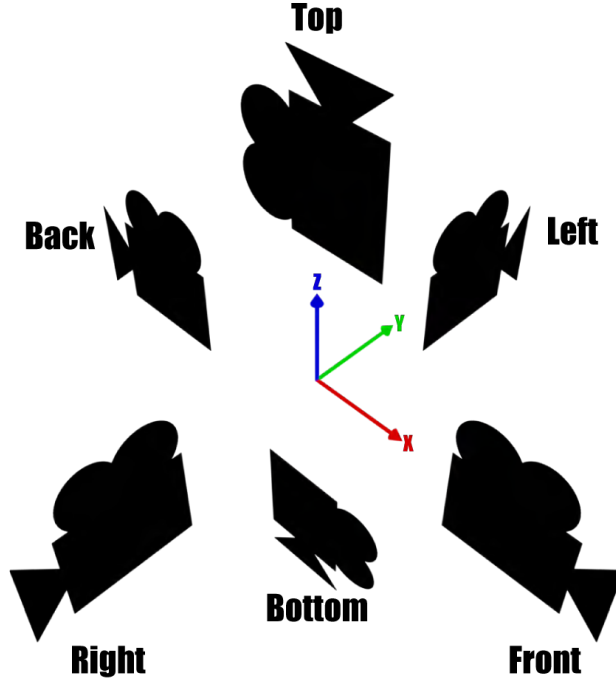


Figure 4.4: Camera cluster

The graphical features captured from all cameras are combined and fed into our indirect lighting predicting neural network. Here, we do the combination for each feature by placing six captured frames in a strategic positioning on a larger frame called a “global view” which is basically a 360° rectangular image. These large frames act as the input for the network. The production of a global view frame is fast because pixels copied to their designated locations. The operation can be represented by an expression:

$$V_{360} = MergeToGlobalView(V_{front}, V_{back}, V_{left}, V_{right}, V_{up}, V_{down}) \quad (4.1)$$

The front view is placed in the middle, the right view slightly to the right, the left view slightly to the left, the back view is sliced into two halves and each slice placed on the far sides. The top and bottom view are deformed and stretched and placed on the top and bottom of the global view respectively, forming a full rectangle.

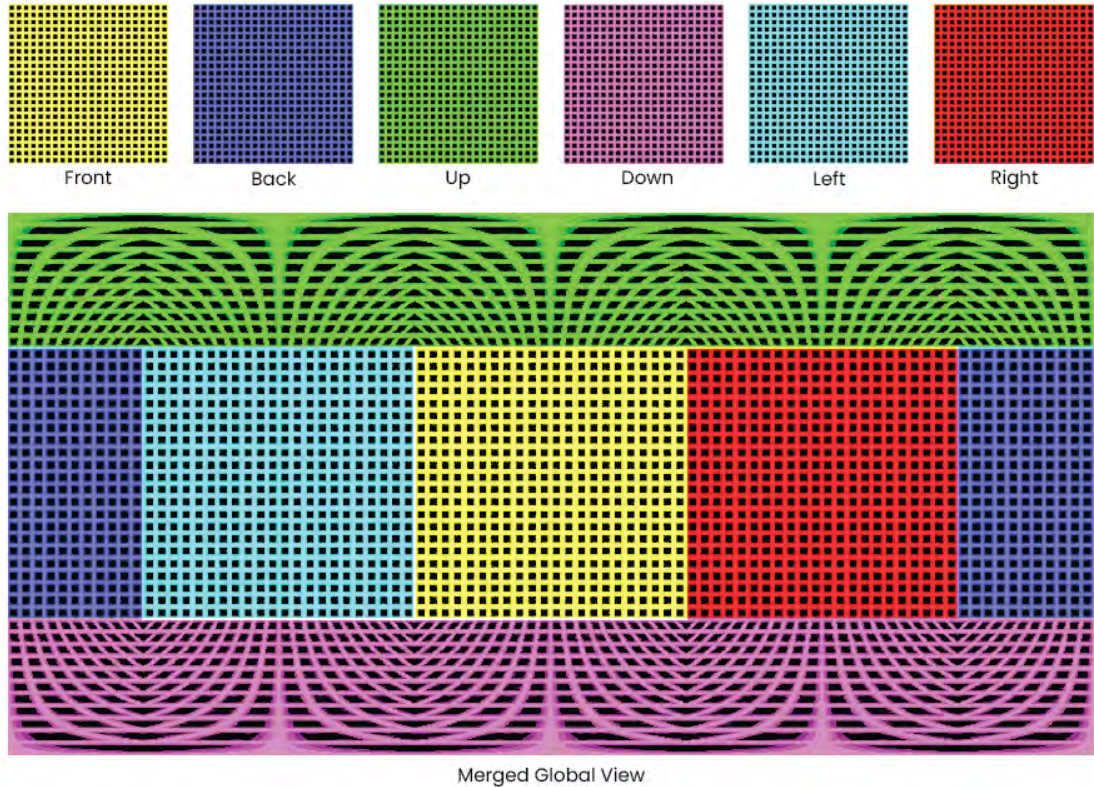


Figure 4.5: Demonstration of the Global View Merging Process

Distortions exist on the up and down portions (the north and south polar regions) of the global view. Only the top and bottom camera views are distorted with duplicate pixels, since it is tough to accommodate all 6 views in a perfect rectangular arrangement. However, this distortion will not be visible since only the front view is presented to the screen, but all views are required for determining the lighting effect from all directions.

The implementable network for this task will be based on Convolutional Neural Network (CNN) in the core. However, a generic CNN-based network does not keep track of location of the kernel, meaning that the convolutional kernel, while scanning the input, does not know in which location of the input it is currently in. For our view-dependent problem, it is crucial to keep track of the direction (here, it corresponds to the location on the global view). Uber AI Labs shows [17] an effortless yet effective solution to this issue by plugging-in two additional channels representing i and j coordinates to an existing CNN-based network, which can take location/direction into account. On top of that, using CoordConv offers faster convergence on training-supervised networks.

For our implementation, we consider the i and j coordinates to be the phi (ϕ) and theta (θ) values of the 3D polar coordinates system when the camera-cluster is placed in the origin. To build the coordinate channels, we place a unit sphere around the camera-cluster in Blender, a popular open-source 3D modeling software. After that, we shade the sphere considering red, green, and blue channels representing Cartesian x , y , and z values, respectively. We render for all views, stitch the views with

the *MergeToGlobalView* function [4.1], and convert the RGB Cartesian values into polar coordinates' phi and theta angles, and normalize them between 0 and 1. We do not consider the radius since it is a unit sphere. We only have to do this once since the coordinate channels are constant across all frames, and we can simply load it from a file prior to applying. In Figure 4.6, the coordinate map is represented as an image where the red channel represents phi and green channel represents theta. Distortions caused by repeated pixels are automatically taken into account throughout the process (noticeable near the poles).

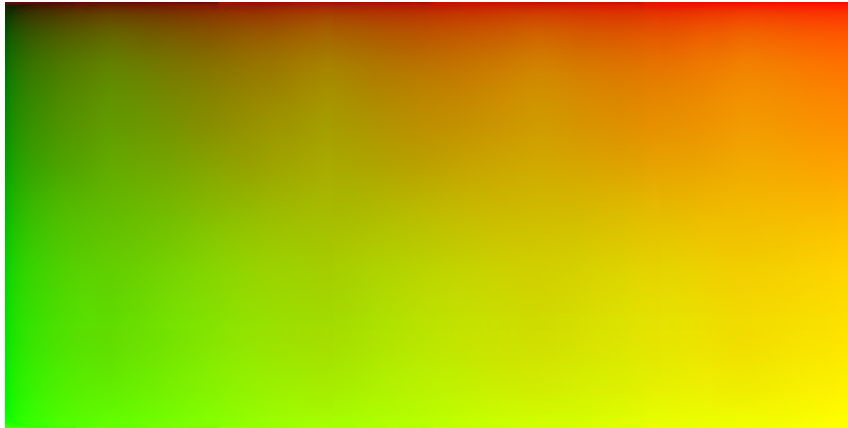


Figure 4.6: Coordinate Map, showing phi (ϕ) (in red) and theta (θ) (in green) coordinate angles.

The whole operation sequence can be summarized by the pseudocode 1

```

Function SurroundingAware():
    flatColorF, normalMapF, depthMapF, lumenMapF ← RenderFrontCamera()
    flatColorB, normalMapB, depthMapB, lumenMapB ← RenderBackCamera()
    flatColorL, normalMapL, depthMapL, lumenMapL ← RenderLeftCamera()
    flatColorR, normalMapR, depthMapR, lumenMapR ← RenderRightCamera()
    flatColorU, normalMapU, depthMapU, lumenMapU ← RenderUpCamera()
    flatColorD, normalMapD, depthMapD, lumenMapD ←
        RenderDownCamera()

    flatColor ← MergeToGlobalView(flatColorF, flatColorB, flatColorL,
        flatColorR, flatColorU, flatColorD)
    normalMap ← MergeToGlobalView(normalMapF, normalMapB,
        normalMapL, normalMapR, normalMapU, normalMapD)
    depthMap ← MergeToGlobalView(depthMapF, depthMapB, depthMapL,
        depthMapR, depthMapU, depthMapD)
    lumenMap ← MergeToGlobalView(lumenMapF, lumenMapB,
        lumenMapL, lumenMapR, lumenMapU, lumenMapD)

    coordMap ← GetCoordinateMap()

    output ← NetworkModel(flatColor, normalMap, depthMap, lumenMap,
        coordMap)
    finallmage ← MultiplyRGB(output, flatColor)

    display(finallmage)
return

```

Algorithm 1: General operation algorithm.

4.2 Dataset Generation

Since the input to the neural network is rendered 3D scenes in the form of feature buffers, we need a diverse dataset consisting of several 3D scenes. Considering our goal, we require a dataset that contains many frames of 3D scenes, both in rendered form, using a ray-tracing based renderer and rasterized images of the same scenes in feature buffer form, and those should cover surrounding view of camera points. Since our requirements are too specific, we could not find any dataset online that covers our needs. So we decided to generate our own dataset for both training and testing purposes.

For fabrication of virtual 3D environments or scenes, we considered using the popular sandboxing game called “Minecraft”. We created some diverse maps in Minecraft and loaded them into a specialized tool called “Mineways” in order to convert these maps to 3D models, so that they can be rendered later according to our settings of choice. Here, we chose a specific portion of the maps and exported the 3D scenes as .obj files. After exporting the files, we imported them into Blender. Inside Blender, we have created the camera-cluster using six cameras for six views (front, back, left, right, up, down), each having a field-of-view of 90 degrees and aspect ratio of 1:1,

where we made the front camera act as the parent for all the remaining cameras. This way, we can ensure that the cameras stay together and travel together as we capture frames from various points within scenes and also maintain their orientation.

Blender has allowed us to modify the lighting conditions and use various post-production effects. We have used daylight for most of the scenes, while we used lighting at night for some of the remaining scenes. A few scenes have dynamic lighting, meaning the lighting changes from daylight to lighting at night, giving the effect of sunset or sunrise. Some scenes are outdoor-focused, and some of them focus on indoor environments.

Instead of placing the camera-cluster in random places within the scenes and rendering manually, we animated the camera-cluster itself to fly through the environments since Blender supports creating 3D animations. We just specified few checkpoints within the environments, placed the camera-cluster in those checkpoints, and recorded the camera location and rotation as keyframes. Each keyframe had an interval of 25 frames. This way, we could effortlessly render all of the frames using a single click and get an animated walkthrough.

For the train dataset, we have prepared 5300 frames from 19 scenes, each having a resolution of 256x256. For the test dataset, we have prepared 1525 frames from 6 different scenes. All the scenes were rendered using Blender’s built-in rasterizer renderer called “EVEE” and built-in ray traced renderer called “Cycles X”. Blender’s compositor allows us to extract feature buffers on a per-frame basis for every camera and save them into individual files. For each frame and each camera view of the cluster, we have collected these feature buffers: diffuse color, direct lighting, depth map, normal map, surface emission map, and environment/sky map. These feature buffers will act as the input to the neural network.

We have also collected ray traced renders of direct and indirect lighting for each frame, but from the viewpoint of the front camera only since we target to predict the lighting only for the front camera view. The blend of direct and indirect lightmaps will work as the ground truth, where the blending function is “mix add”. It may seem redundant to collect direct lightmaps again for ray tracing as we have already collected them for rasterization earlier. However, Blender’s ray-traced direct lightmaps contain some extra lighting features, such as environment scatter lights and lights from self-emitting objects, which are not available in rasterization.

The diffuse color, direct lighting, indirect lighting, surface emission map, and environment/sky map images, each consist of 3 color channels: red, green, and blue. The normal map images each consist of 3 axis-vector channels: X, Y, and Z. The depth map images each consist of 1 scalar channel which is the depth itself. We saved the diffuse color, direct lighting, indirect lighting, surface emission map, and environment/sky map images as 8-bit per channel RGB png images, the normal map as 32-bit per channel RGB exr images, and the depth map as 32-bit per channel single channel exr images. In addition to the images, we have also collected the orientation data of the camera cluster for each frame. This orientation data consists of the look vector, and up vector of the front camera, which will help in the prepro-

cessing phase later. The extraction is performed by running a Python script within Blender’s internal Python interpreter. The script iterates over all frames, reads the camera orientation-vector values, and stores them into an array, which is saved as JSON files. This process is done for each of the scenes.

	Back	Down	Front	Left	Right	Up
Depth						
Normal						
Diffused						
Direct						
Emission						
Environment						

Figure 4.7: Sample feature frame from the dataset.

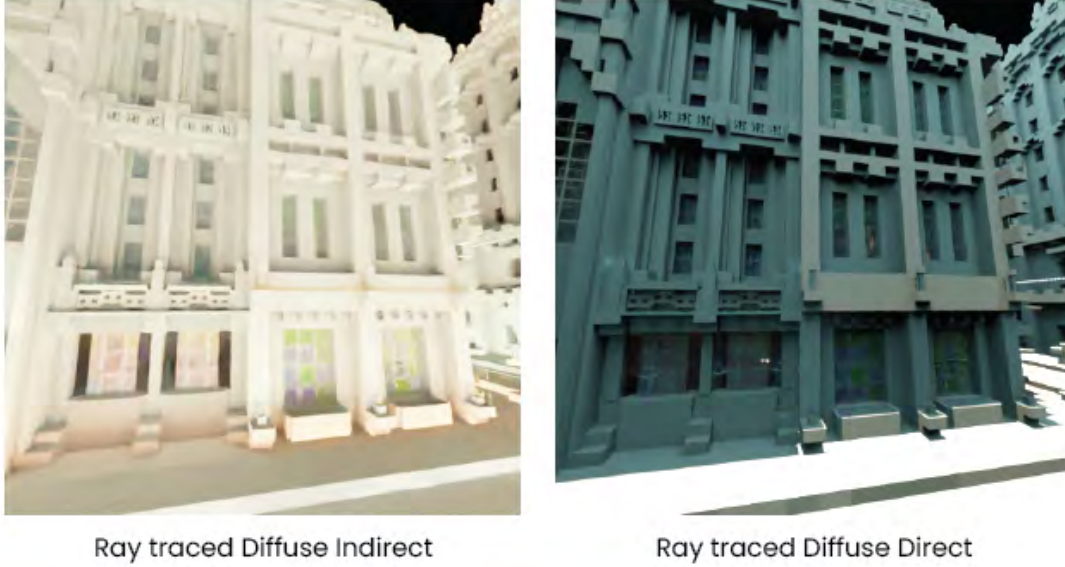


Figure 4.8: Ground truth sample of a frame from the dataset

4.3 Dataset Pre-processing

The dataset is complete but not ready for the network to consume yet, mostly because the images are in formats and shapes that are incompatible with our targeted network models. Blender outputs captured images of feature buffers and renders of each camera into separate files, and saves vector space information like depth information and normals in raw unscaled values. To make the data compatible for our network models, we must pre-process every frame into a suitable form. We perform this task in two stages. The first stage is mainly concerned with transforming the pixel values themselves. And in the second stage, the feature buffers are stitched to form global views for surrounding-aware input.

4.3.1 Phase 1

In the first preprocessing stage, we keep the diffuse color maps untouched. The depth map values are normalized to 0 and 1, since blender stored the depth values in meters. The max distance value considered is 1,000 meters, any distance above that limit is clamped to 1 automatically. The direct lightmap, emissive lightmap, and the environment lightmaps are all combined into one feature map called the lumination map, since all 3 can be considered as light sources for determining indirect light. The combination is done by per-pixel addition of RGB values.

$$\lambda_{lum} = \lambda_{emit} + \lambda_{env} + \lambda_{direct} \quad (4.2)$$

Here, λ_{lum} , λ_{emit} , λ_{env} , λ_{direct} corresponds to lumination lightmap, emissive lightmap, environment lightmap, direct lightmap respectively.

Blender’s compositor provides surface normals in world space, which may appear too complex for the neural network. We convert the world space data to camera

space, meaning the normal vectors will follow the camera-cluster's own alignment axis as their basis. It is done by basis transformation, and we utilize the previously recorded per-frame look and up vectors of the camera. We calculate the basis for each frame using these two vectors, represent the basis in a 3x3 matrix (S), and considering the normal maps' of each pixel's RGB values as XYZ of the normal vector (\vec{n}), we multiply the normal vectors with the pre-calculated matrix, which gives us the per-pixel camera space normal (\vec{N}). Afterwards, since the vector values are between -1 and 1, we normalize them between 0 and 1.

$$\hat{\beta} = up_x \hat{i} + up_y \hat{j} + up_z \hat{k} \quad (4.3)$$

$$\hat{\gamma} = -look_x \hat{i} - look_y \hat{j} - look_z \hat{k} \quad (4.4)$$

$$\hat{\alpha} = \hat{\beta} \times \hat{\gamma} \quad (4.5)$$

$$S = \begin{bmatrix} \alpha_x & \alpha_y & \alpha_z \\ \beta_x & \beta_y & \beta_z \\ \gamma_x & \gamma_y & \gamma_z \end{bmatrix} \quad (4.6)$$

$$\vec{n} = \begin{bmatrix} n_x \\ n_y \\ n_z \end{bmatrix} \quad (4.7)$$

$$\vec{N} = S \times \vec{n} \quad (4.8)$$

	Back	Down	Front	Left	Right	Up
Depth						
Normal						
Diffused						
Lumination						

Figure 4.9: Sample frame from the 1st preprocessing stage output

For the ray traced direct lightmap and indirect lightmap, we simply blend their per-pixel RGB values using the mix-add blending function into a new lightmap called the ground truth lumination. The neural network will try to generate this image by consuming the features. The generated ground truth images will be used to train the network and verify its accuracy.

$$\Lambda_{lum} = \Lambda_{direct} + \Lambda_{indirect} \quad (4.9)$$

Here, Λ_{lum} , Λ_{direct} and $\Lambda_{indirect}$ represents ground truth lumination values, direct lightmap values and indirect lightmap values respectively.

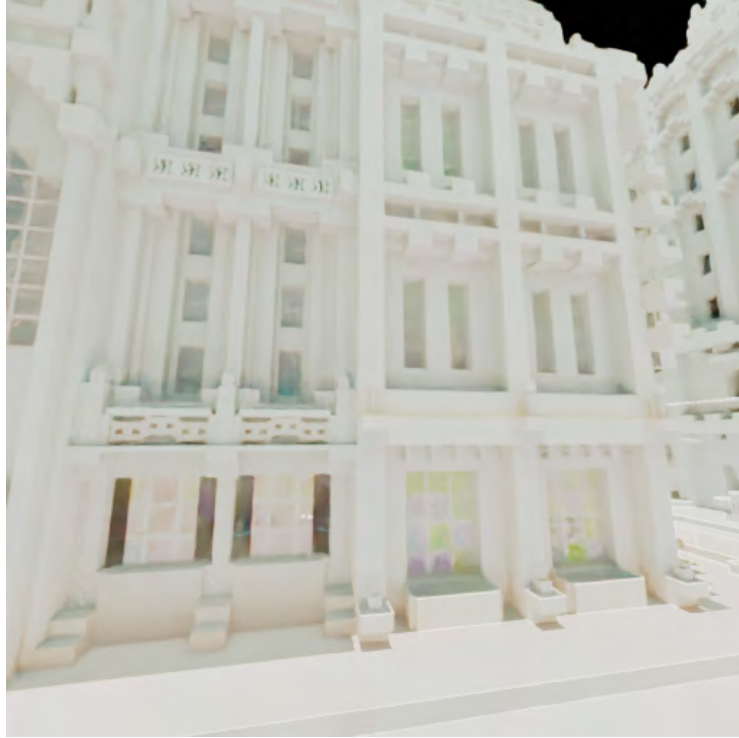


Figure 4.10: Ground Truth Lumination

4.3.2 Phase 2

In this phase of preprocessing, we combine all six view images from six cameras into one big 360° image called the global view, which is done for all of the 4 features. For this process, we utilize the *MergeToGlobalView* function [4.1], which takes 6 images (of each view) and outputs one rectangular image. For our experiment, this rectangular image has a resolution of 1024x512 pixels. Nothing needs to be done for the ground truth lumination since we only target to generate lumination data for the front view.





<p>Depth</p>	
<p>Diffuse Color</p>	
<p>Lumination</p>	
<p>Normals</p>	

Figure 4.11: Sample frame from the 2nd preprocessing stage output

4.4 Model Architecture

The architecture we follow for our lighting prediction system is a Generative Adversarial Network (GAN). To be more specific, it is a conditional GAN (cGAN), where the output image of the network depends on the input image [13]. It is a combination of Generator network and Discriminator network [7]. The generator consumes the rasterized feature buffers, along with the coordinate maps, and tries to generate a possible output image, regarding it as a synthetic image. The discriminator

takes either the synthetic output or the ground truth image and also the respective feature buffers, it learns to identify the real ground truth images from the synthetic one based on the feature buffers. While adjusting the parameters during training, the parameters of the discriminator are updated directly, and the parameters of the generator are updated through the discriminator. However, we also employ L1 loss to update generator parameters, which are calculated by comparing the synthetic image and ground truth. As mentioned by Image-to-Image Translation with Conditional Adversarial Networks [13], combining both GAN loss and L1 loss yields better accuracy. For the combined GAN model, BCE (Binary Cross Entropy) loss is employed, which is a part of the adversarial loss. The total generator is calculated by using equation 4.10, where $\lambda = 100$

$$\text{generator loss} = \text{adversarial loss} + \lambda * \text{L1 loss} \quad (4.10)$$

In this implementation, the generator follows a U-Net architecture [10], a combination of an encoder and a decoder. The encoder part gradually downsamples inputs and congregates information into a narrow layer. The decoder gradually upscales the congregated information, while also concatenating information via skip connections from corresponding encoder layers [13]. In the encoder part, it maintains series blocks of Convolution, Batch Normalization, and LeakyReLU [6] layers of count 8 with Batch Normalization being absent in the first layer. While the decoder has a series of blocks of Convolutional Transpose, Batch Normalization, Dropout, and ReLU layers also of count 8 with the dropout rate set to 50%. The centermost block, also known as the bottleneck block is free from any Batch Normalization layer as activations there are zeroed out due to the Batch Normalization operation `pix2pix`. The end of the network is the output featuring 3 channels for RGB, activated by the Tanh activation function. The discriminator is similar to the encoder portion of the generator, but it follows a patchGAN architecture [13]. It penalizes the generator based on individual patches and outputs a realness score. It contains a series of blocks of Convolution, Batch Normalization, and LeakyReLU layers of count 5 with Batch Normalization being absent in the first layer. The last layer delivers a 1-dimensional output, activated by a sigmoid activation function [13]. We use a base kernel count of 64 for both networks, the intermediary layers having kernel count multiple of the base count. The generator takes the feature buffers and coordinate map as input, with channels from all features, concatenated to a total of 12 channels and produces an output RGB image of 3-channels. The discriminator takes all the 12 channels the generator takes, concatenated with a 3-channel image (either generated image or ground truth), a total of 15 channels, and outputs a probability of realism of the image. In generator, we have 48 layers which are demonstrated in Table 4.1 and Table 4.2 respectively. For discriminator, it contains 13 layers which are visualized in Table 4.4.

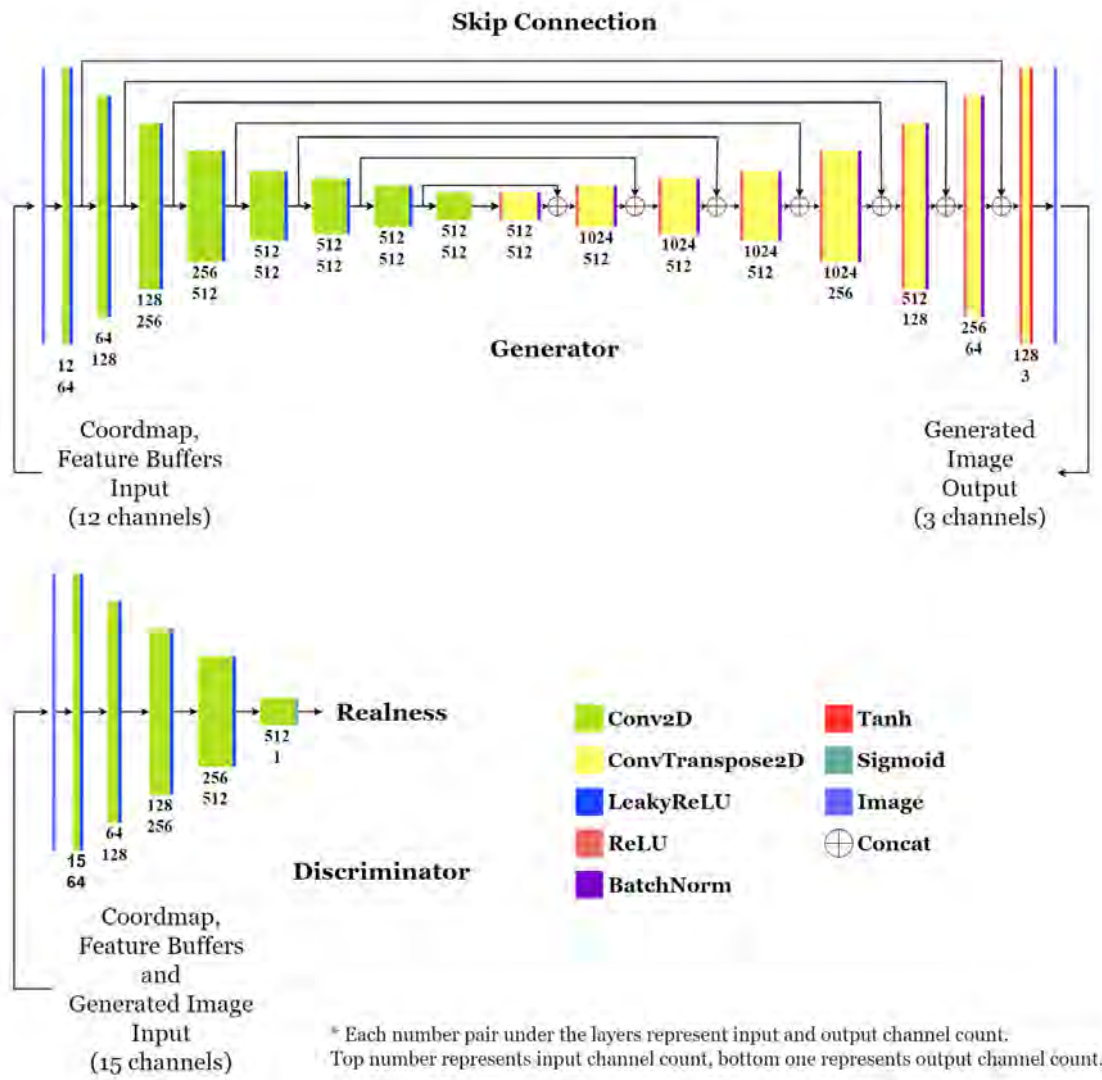


Figure 4.12: First image represents the generator model and second image represents the discriminator model

#	Layer Name	Shape	Parameter Count
1	Conv2d	[1, 64, 256, 512]	12,352
2	LeakyReLU	[1, 64, 256, 512]	None
3	Conv2d	[1, 128, 128, 256]	131,200
4	BatchNorm2d	[1, 128, 128, 256]	256
5	LeakyReLU	[1, 128, 128, 256]	None
6	Conv2d	[1, 256, 64, 128]	524,544
7	BatchNorm2d	[1, 256, 64, 128]	512
8	LeakyReLU	[1, 256, 64, 128]	None
9	Conv2d	[1, 512, 32, 64]	2,097,664
10	BatchNorm2d	[1, 512, 32, 64]	1,024
11	LeakyReLU	[1, 512, 32, 64]	None
12	Conv2d	[1, 512, 16, 32]	4,194,816
13	BatchNorm2d	[1, 512, 16, 32]	(recursive)
14	LeakyReLU	[1, 512, 16, 32]	None
15	Conv2d	[1, 512, 8, 16]	4,194,816
16	BatchNorm2d	[1, 512, 8, 16]	(recursive)
17	LeakyReLU	[1, 512, 8, 16]	None
18	Conv2d	[1, 512, 4, 8]	4,194,816
19	BatchNorm2d	[1, 512, 4, 8]	(recursive)
20	LeakyReLU	[1, 512, 4, 8]	None
21	Conv2d	[1, 512, 2, 4]	4,194,816
22	ReLU	[1, 512, 2, 4]	None
23	ConvTranspose2d	[1, 512, 4, 8]	4,194,816
24	BatchNorm2d	[1, 512, 4, 8]	(recursive)

Table 4.1: First 24 layers of Generator

#	Layer Name	Shape	Parameter Count
25	Dropout	[1, 512, 4, 8]	None
26	ReLU	[1, 1024, 4, 8]	None
27	ConvTranspose2d	[1, 512, 8, 16]	8,389,120
28	BatchNorm2d	[1, 512, 8, 16]	(recursive)
29	Dropout	[1, 512, 8, 16]	None
30	ReLU	[1, 1024, 8, 16]	None
31	ConvTranspose2d	[1, 512, 16, 32]	8,389,120
32	BatchNorm2d	[1, 512, 16, 32]	(recursive)
33	Dropout	[1, 512, 16, 32]	None
34	ReLU	[1, 1024, 16, 32]	None
35	ConvTranspose2d	[1, 512, 32, 64]	8,389,120
36	BatchNorm2d	[1, 512, 32, 64]	(recursive)
37	ReLU	[1, 1024, 32, 64]	None
38	ConvTranspose2d	[1, 256, 64, 128]	4,194,560
39	BatchNorm2d	[1, 256, 64, 128]	(recursive)
40	ReLU	[1, 512, 64, 128]	None
41	ConvTranspose2d	[1, 128, 128, 256]	1,048,704
42	BatchNorm2d	[1, 128, 128, 256]	(recursive)
43	ReLU	[1, 256, 128, 256]	None
44	ConvTranspose2d	[1, 64, 256, 512]	262,208
45	BatchNorm2d	[1, 64, 256, 512]	128
46	ReLU	[1, 128, 256, 512]	None
47	ConvTranspose2d	[1, 3, 512, 1024]	6,147
48	Tanh	[1, 3, 512, 1024]	None

Table 4.2: Remaining layers of Generator

Total params	54,420,739
Trainable params	54,420,739
Non-trainable parameters	0

Table 4.3: Parameter count of generator model

#	Layer Name	Shape	Parameter Count
1	Conv2d	[1, 64, 256, 512]	15,424
2	LeakyReLU	[1, 64, 256, 512]	None
3	Conv2d	[1, 128, 128, 256]	131,200
4	BatchNorm2d	[1, 128, 128, 256]	256
5	LeakyReLU	[1, 128, 128, 256]	None
6	Conv2d	[1, 256, 64, 128]	524,544
7	BatchNorm2d	[1, 256, 64, 128]	512
8	LeakyReLU	[1, 256, 64, 128]	None
9	Conv2d	[1, 512, 63, 127]	2,097,664
10	BatchNorm2d	[1, 512, 63, 127]	1,024
11	LeakyReLU	[1, 512, 63, 127]	None
12	Conv2d	[1, 1, 62, 126]	8,193
13	Sigmoid	[1, 1, 62, 126]	None

Table 4.4: Layers of Discriminator

Total params	2,778,817
Trainable params	2,778,817
Non-trainable parameters	0

Table 4.5: Parameter count of discriminator model

4.5 Training and Testing

We have decided to build our model and have build another model that produces screen-space global illumination using data from a single camera based on DeepIllu- mination’s [15] architecture to perform a side-by-side comparison. We prepared all our networks using PyTorch [21] for both training and testing purposes. Training is done for 40 epochs on both the architectures, based on monitoring the mathematical and visual improvement of the outputs. We utilized the benefits of hardware accel- erated training using GPUs, specifically PyTorch’s CUDA support. The GPUs we used were Nvidia GeForce GTX 1660 Ti and Nvidia GeForce RTX 3060 ELITE for local training, and Nvidia Tesla T4 and Nvidia Tesla K80 for training on the cloud, provided by Google Colab. According to Image-to-Image Translation with Conditional Adversarial Networks [13], a batch size of 1 yields better results for a U-Net-based generator, so we kept the frame count to 1 per batch. For all the models, we maintained a learning rate of 0.0002 and a dropout rate of 0.5, which protects us from over-fitting the models [13]. While training, we added the functionality to dump generator output as images to inspect them manually while the training process goes on. Besides that, the models were saved as files after ev-

ery epoch, allowing us to use them for future use cases such as testing and evaluation.

The same dataset is used for both the model using our Surrounding-Aware model and the DeepIllumination model, to perform impartial training and testing. Since the DeepIllumination model can take feature buffers from only one camera, only data from the front camera is fed into it. While testing, we fed samples from the test dataset to the generator outputs a fabricated lightmap/lumination map that contains global illumination. Lightmaps are generated for both architectures. Those light maps are compared against the ground-truth lightmap/lumination map for metrics calculation. Just like the training phase, the images are dumped from memory and saved as files for future manual analysis. The training process can be represented by the pseudocode [2]

```

LAMBDA ← 100
G ← CreateGenerator()
D ← CreateDiscriminator()

Function TrainBatch(flatColor, normalMap, depthMap, lumenMap,
groundTruth):
    coordMap ← GetCoordinateMap()

    disScoreReal ← D(flatColor, normalMap, depthMap, lumenMap, coordMap,
groundTruth)
    disLossReal ← BCELoss(disScoreReal, 1)

    backPropagation(disLossReal)

    fakedImage ← G(flatColor, normalMap, depthMap, lumenMap, coordMap)

    disScoreFake ← D(flatColor, normalMap, depthMap, lumenMap,
coordMap, fakedImage)
    disLossFake ← BCELoss(disScoreFake, 0)

    backPropagation(disLossFake)
    disScoreGen ← D(flatColor, normalMap, depthMap, lumenMap, coordMap,
fakedImage)
    genLoss ← BCELoss(disScoreGen, 1)
    genLossLi ← LiLoss(fakedImage, groundTruth)
    genLoss ← genLoss + LAMBDA * genLossLi
    backPropagation(genLoss)
return

Function TrainEpochs(epochCount, dataset):
    for i to epochCount do
        for j to datasetLength do
            flatColor, normalMap, depthMap, lumenMap, groundTruth ←
LoadSample()
            TrainBatch(flatColor, normalMap, depthMap, lumenMap,
groundTruth)
        end
    end

    SaveModel(G)
    SaveModel(D)
return

```

Algorithm 2: Training process algorithm.

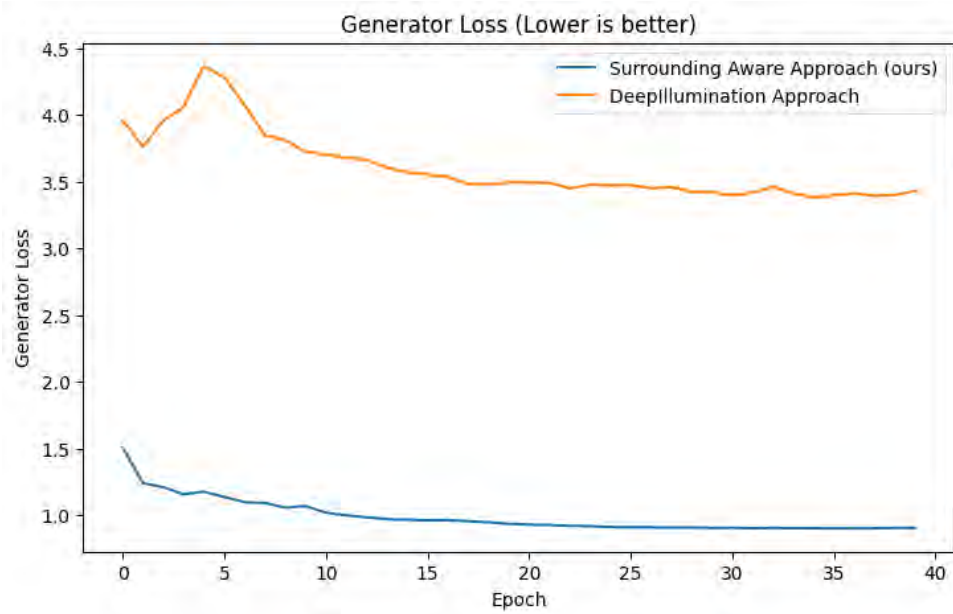


Figure 4.13: Generator loss.

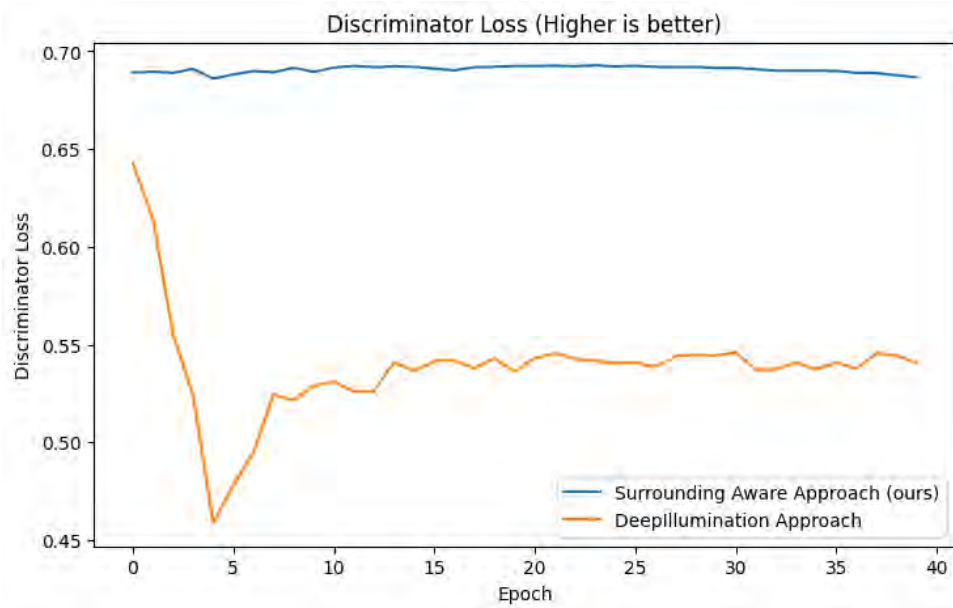


Figure 4.14: Discriminator loss.

Chapter 5

Performance Evaluation

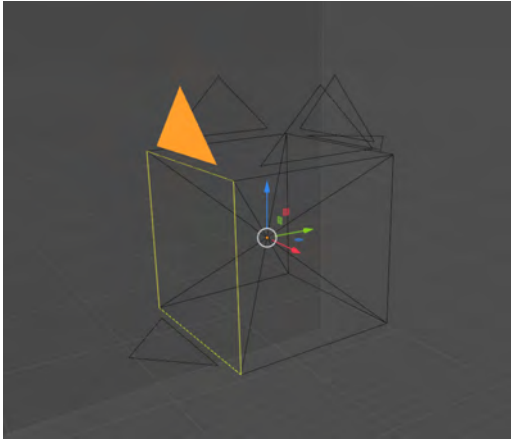
In this chapter, we will be discussing about the experimental configurations, experimental results, and the experimental findings of our thesis. In our experimental configurations, we have discussed how we setup our environment for dataset generation. After that we have elucidated our training configuration and testing configuration. In our experimental results, we have demonstrated our result output comparison, visual consistency analysis, quality evaluation metrics, and performance evaluation metrics. Finally, in our experimental findings, we have explained our findings from quality evaluation metrics and time performance. Afterwards, we have done a theoretical comparison between our approach and other existing approaches.

5.1 Experimental Configurations

We have set up hardware and software environments suitable for our three major tasks: dataset generation, training the networks, and testing or evaluating the networks. All of these tasks are hardware accelerated, meaning each will utilize the GPUs.

5.1.1 Dataset Generation

For generating the dataset, we have used Blender version 3. The camera cluster was placed within the previously created 3D environments, and “multi-view” rendering was kept enabled. In Blender’s compositor section, nodes were set up to dump feature buffers as files. We render each scene two times, one with rendering engine set to “Cycles X”, and another one set to “EVEE”. We ran the “Cycles X” renders on a machine equipped with a NVIDIA GeForce RTX 3060 ELITE GPU due to its hardware accelerated ray-tracing feature.



(a) Camcluster setup in Blender (Front camera highlighted)



(b) Camcluster hierarchy in Blender

The camera cluster (camcluster) is made using arranging six cameras in a cube-like manner, each of them having an adjacent angle between them of 90° . The front camera is made the parent of all other remaining cameras.



Figure 5.2: High-level compositor node setup for feature buffers

During rendering using EEVEE, or the rasterizer, we capture depth maps (Depth), normal maps (Normal), diffuse color or flat color (DiffCol), direct light maps (DiffDir), emission maps (Emit), and environment lights or sky lights (Env) from the Render Layers for every camera. These correspond to the feature buffers, which are sent to a custom node group for further processing.

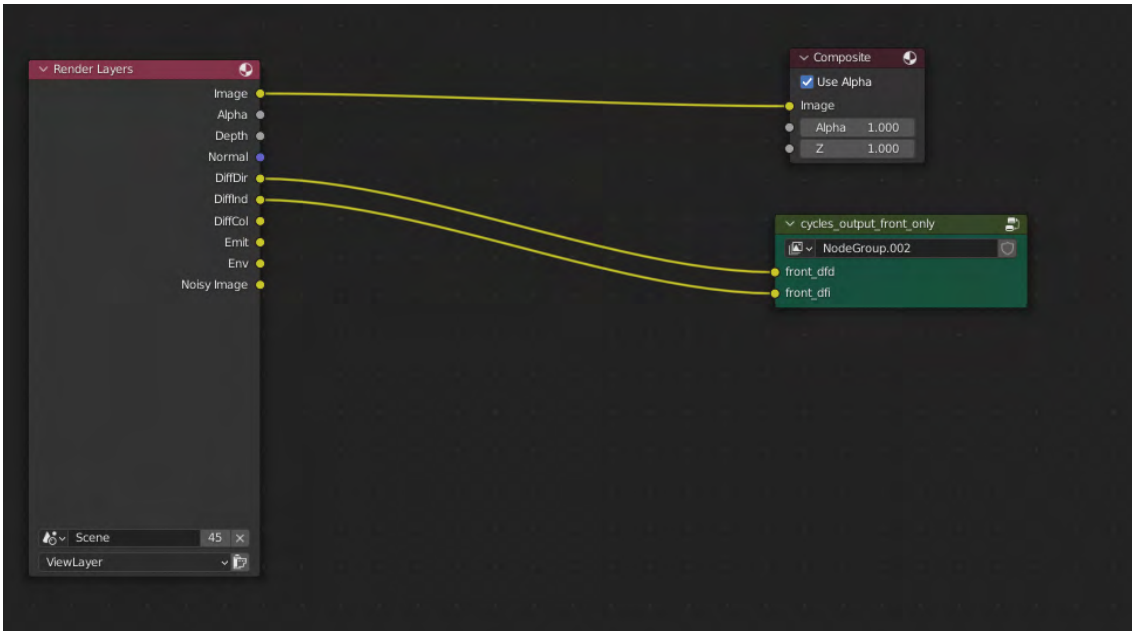


Figure 5.3: High-level compositor node setup for ground truth

During rendering using Cycles X, or the ray-tracer, we capture direct light map (DiffDir) and indirect light map (DiffInd) from the Render Layers for only the front camera. These correspond to the ground truth, which is sent to a custom node group for further processing.

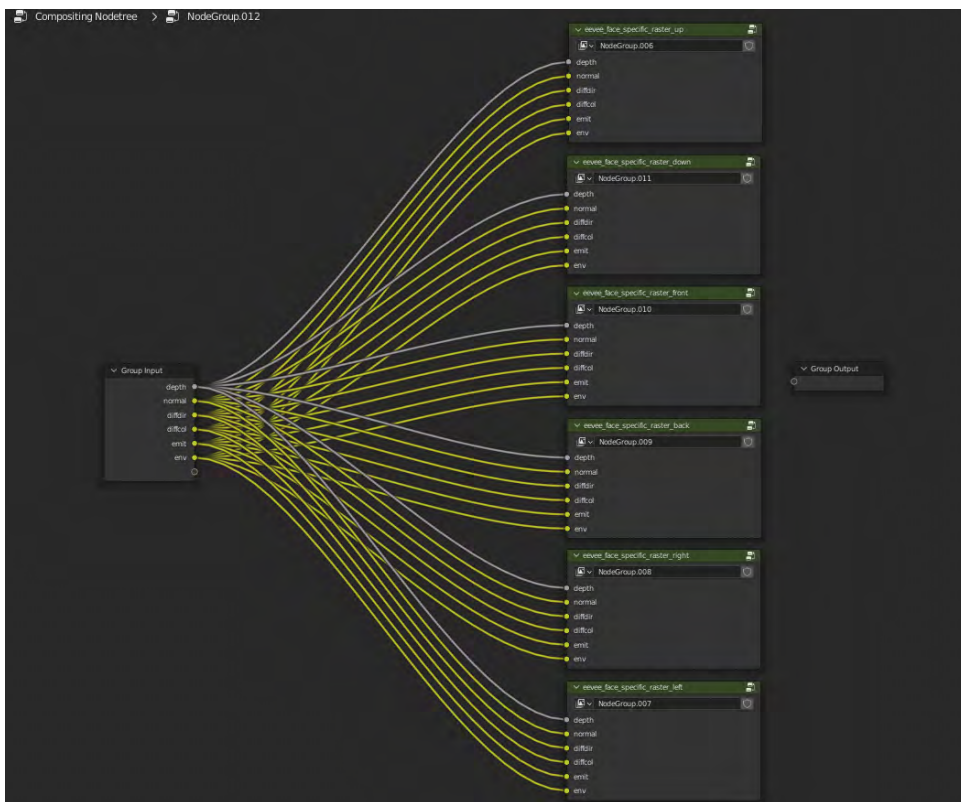


Figure 5.4: Compositor nodes for feature buffer view routing

Feature buffers from each camera need to be separated via “Switch View” nodes, so

the combined feature buffers are sent to individual custom file saver nodes dedicated to each camera/face.

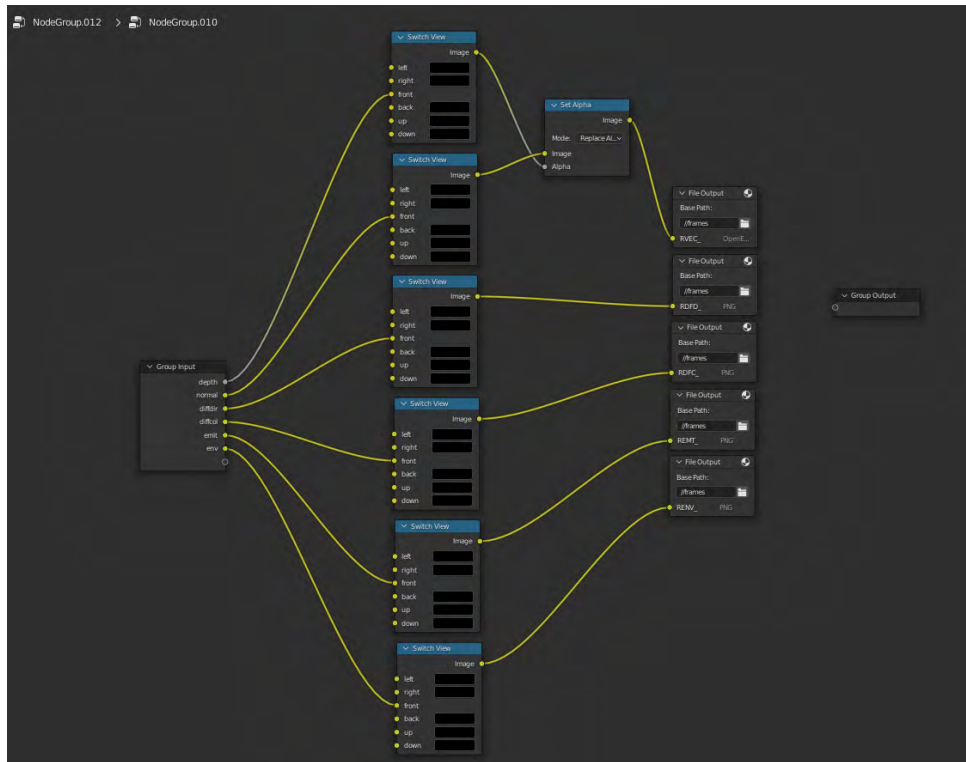


Figure 5.5: Compositor nodes saving feature buffer to file (shown for front camera)

The custom file saver node filters out only the feature intended for it, depending on which camera it corresponds to. The depth maps and normal maps are combined to be saved as a 32-bit per channel RGBA exr, while every other image is saved as 8-bit per channel RGB images.

5.1.2 Training Configuration

For experimenting, we have used our local machines and Google Colab. Training and testing were done in separate machines during the experimental phase. For training the model, we have used NVIDIA GeForce RTX 3060 ELITE, NVIDIA Geforce GTX 1660 Ti and NVIDIA Tesla T4 from Google Colab, all with CUDA enabled. During the testing phase, we have tested our trained model using Geforce GTX 1660 Ti and Geforce GTX 1050 Ti. The detailed specifications for each including test benches are

Components	Device-1 (RTX 3060 ELITE)	Device-2 (GTX 1660 Ti)	Device-3 Google Colab
Processor	AMD Ryzen 5 3600 CPU @ 3.60GHz	AMD Ryzen 5 3600 CPU @ 3.60GHz	Intel(R) Xeon(R) CPU @ 2.20GHz
GPU	NVIDIA GeForce RTX 3060 ELITE	NVIDIA GeForce GTX 1660 Ti	Nvidia Tesla T4
Clock speed	3.60GHz	3.60GHz	2.20GHz
RAM	16GB	16GB	12GB

Table 5.1: Hardware configuration used during training phase

5.1.3 Testing Configuration

For testing, only the generator model is loaded into the memory. Just like the testing configuration, acceleration using CUDA is also enabled here. We chose to use two medium range non-RTX GPUs, NVIDIA Geforce GTX 1660 Ti and NVIDIA Geforce GTX 1050 Ti, to run the test model. The detailed specifications for each, including test benches, are

Components	Device-1 (GTX 1050 Ti)	Device-2 (GTX 1660 Ti)
Processor	Intel(R) Core(TM) i5-7400 CPU @ 3.00GHz	AMD Ryzen 5 3600 CPU @ 3.60GHz
GPU	NVIDIA GeForce GTX 1050 Ti	NVIDIA GeForce GTX 1660 Ti
Clock speed	3.00GHz	3.60GHz
RAM	16GB	16GB

Table 5.2: Hardware configuration used during testing phase

5.2 Experimental Results

For evaluation and application purposes, we only need the trained generator model since it is the only part participating in producing global illumination lightmaps. The generator model can be called just like a simple function call by passing the feature-buffer image tensors as parameters. The return value is the resulting global-illumination lightmap image tensor. This image tensor can be compared with the ground-truth lightmap image tensor to calculate the metrics. Or if used in application, it (the generated image) can be blended with the flat-color/albedo image to result the final image. However, it need not be the final image of the graphics pipeline, as it can act as an input to further post-processing stages.

5.2.1 Render Output Comparison

For output comparison, we present a side-by-side comparison of sample frames each containing a rasterized render using direct light only, another rasterized render but

with our network generated global illumination lightmap blended, and a reference image which is the ray-traced ground truth.

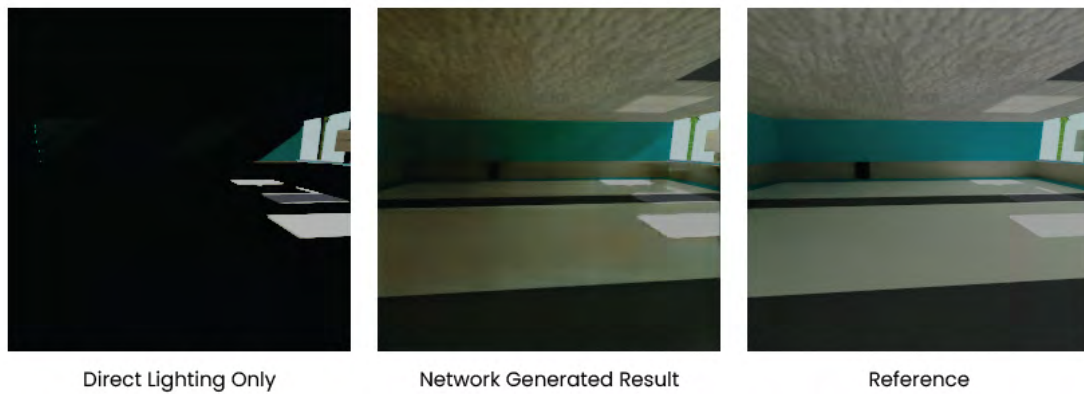


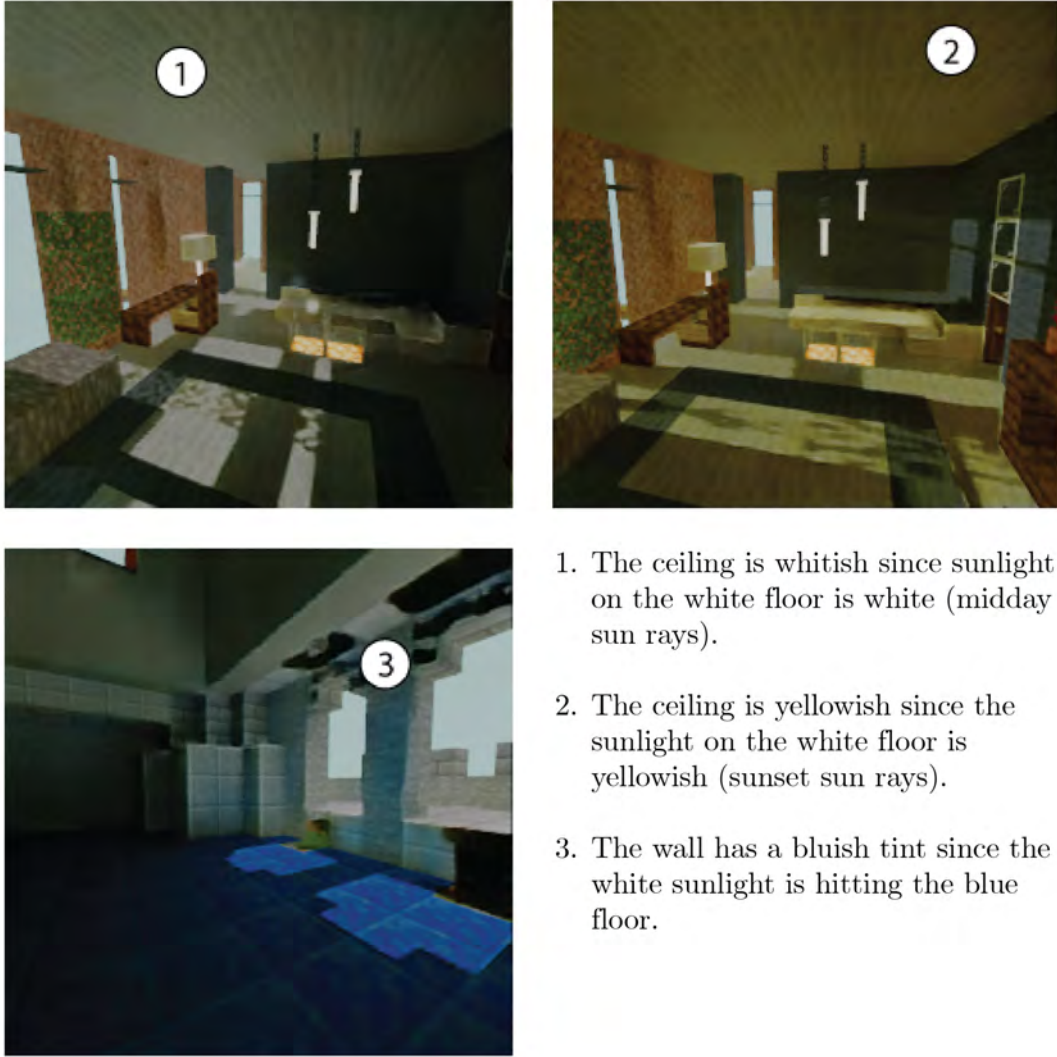
Figure 5.6: The sunlight that hits the floor gets scattered multiple times within the room. (The hole in the wall is darker due to less light rays bouncing there.)



Figure 5.7: A scene mostly illuminated by sky scattering rather than direct sunlight.



Figure 5.8: The surfaces directly lit by sunlight are behind the camera, yet the surfaces visible by the main camera are luminated by indirect light.



1. The ceiling is whitish since sunlight on the white floor is white (midday sun rays).
2. The ceiling is yellowish since the sunlight on the white floor is yellowish (sunset sun rays).
3. The wall has a bluish tint since the white sunlight is hitting the blue floor.

Figure 5.9: Network generated result. Shows that the generated indirect lighting is dependent on diffuse surface color and direct color.

5.2.2 Visual Consistency Analysis

As we have dumped global illumination lightmap outputs produced by the generator network during the testing phase, we have also done a side-by-side comparison between the two aforementioned approaches. The next two pages show a sequence of generated animation frames from two different scenarios to compare the inter-frame lumination consistency. All the generated frames shown are raw output, meaning that those have not been blended with flat-color image, producing the final presentable image on the screen.



Figure 5.10: Moving camera, stationary light. For each pair, top is DeepIllumination [15] approach, bottom is Surrounding-Aware approach (ours)

The animation sequence (Figure 5.10) shows the camera moving into the room while the light is stationary. Notice that in the DeepIllumination [15] approach the luminance inside the room gradually gets dimmer or off despite the light remaining constant due to the directly lit surfaces going out of the screen, eventually losing consistency. While in the Surrounding-Aware approach, the indoor luminance keeps its consistency since it can still keep information about the lit surfaces that do not appear in the viewport.



Figure 5.11: Stationary camera, moving light. For each pair, top is DeepIllumination [15] approach, bottom is Surrounding-Aware approach (ours)

The animation sequence (Figure 5.11) shows an indoor environment of a room where light gradually enters from a window at the back while the camera orientation remains stationary. This emulates an animation of a sunset where the sunlight hits the floor. In the DeepIllumination [15] approach, the luminance remains constant despite the sunlight entering the room of the screen and responds to change or gets brighter when enough lit surface area is present in the viewport. While in the Surrounding aware approach, the room initially remains dark as there is no light in the beginning, but gradually gets brighter as more light enters behind the viewport.

5.2.3 Quality Evaluation Metrics

To evaluate the metrics, we need to examine the image quality using mathematical approach. Since, we are evaluating global illumination, it is needed to assess the locational pixel brightness difference and overall structural similarity of generated images, tested against the ray-traced ground truth. So, we measured our models' output quality accuracy using the common metrics used for global-illumination technique evaluation [8]. These are Mean Squared Error (MSE) and Structural-Similarity Index Measure (SSIM). We also calculated the Peak Signal-To-Noise Ratio (PSNR) for output samples. We have recorded these measurements for every epochs' generated output on the test dataset.

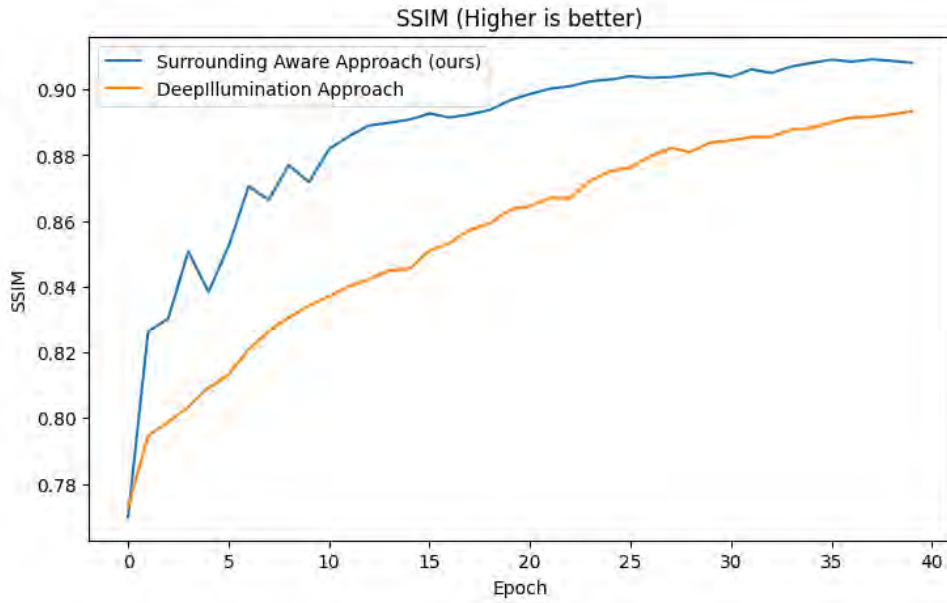


Figure 5.12: Structural-Similarity Index Measure.

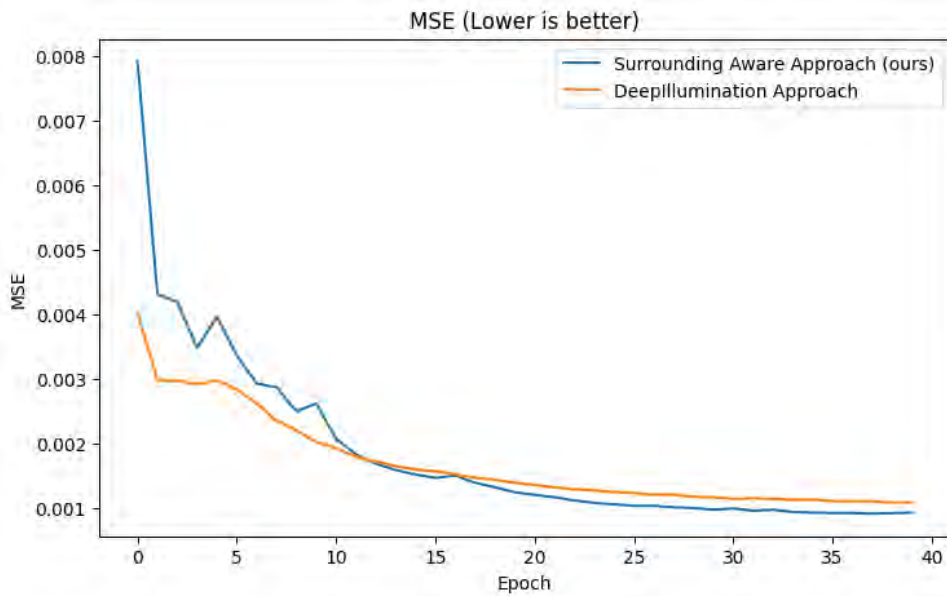


Figure 5.13: Mean Squared Error.

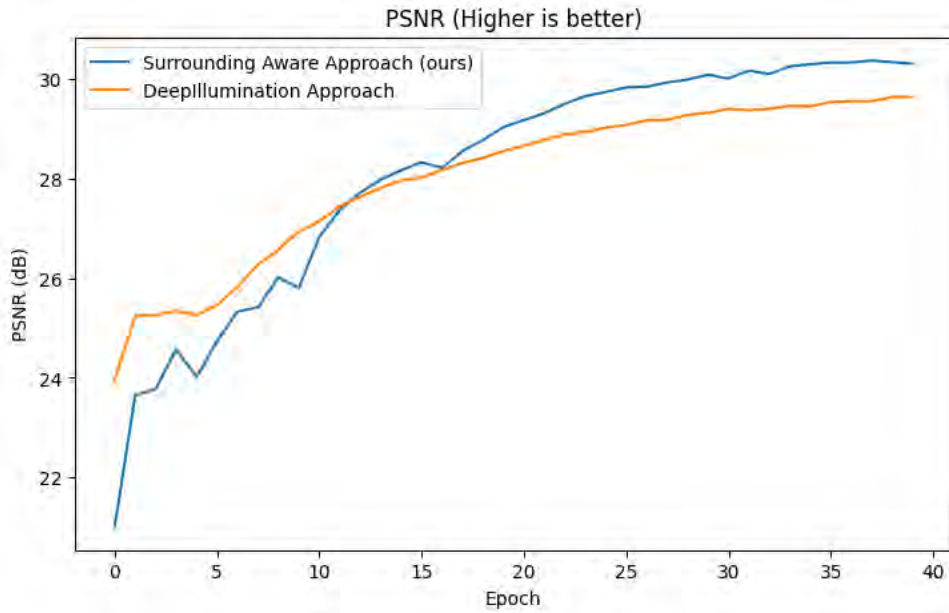


Figure 5.14: Peak Signal-to-Noise ratio.

Approach	SSIM	MSE	PSNR (dB)
Surrounding-Aware (ours)	0.90811	0.00093	30.30576
DeepIllumination [15]	0.89329	0.00108	29.63102

Table 5.3: Quality Evaluation Metrics comparison at 40th epoch.

5.2.4 Performance Evaluation Metrics

The aforementioned table presents the comparison of time taken between different rendering approaches.

GPU	DeepIllumination [15]	Surrounding Aware	Ray traced (Blender)
Nvidia Geforce GTX 1660 Ti	3.3221 ms	3.3196 ms	6.77 s
Nvidia GeForce GTX 1050 Ti	3.7817 ms	4.0210 ms	14.34 s

Table 5.4: Time Performance comparison

5.3 Experimental Findings

Here we discuss the findings upon analyzing the results.

5.3.1 Findings About Quality Evaluation Metrics

According to the metrics comparison between Surrounding-Aware approach (ours) and DeepIllumination [15] approach, our approach outperforms the other one in all 3 metrics. For SSIM and PSNR, a higher value is desired. Whereas in the case of

MSE, a lower value means better quality of generated output. This shows providing enough surrounding details to a global illumination producing GAN yields better accuracy and closer to ground-truth (ray-traced) results.

5.3.2 Findings About Time Performance

The ray-traced render times for both hardware came out to be unsuitable for real-time usage. However, both Surrounding-Aware and DeepIllumination [15] processing time of 3-4 milliseconds came out to be good for usage in interactive applications. In most hardware, the DeepIllumination [15] implementation would perform better due to its input size being comparatively smaller but considering its lacking, judging by the visual and mathematical metrics comparison, against our approach, it would be better to use the Surrounding-Aware system instead. To be fair, the performance difference between the two is not much. And from our observation, our Surrounding-Aware approach performed better than DeepIllumination [15] approach on the “Nvidia Geforce GTX 1660 Ti” hardware.

5.3.3 Theoretical Comparison

This section compares our implementation with similar neural network related technologies those generate realistic effects for real-time rasterized graphics via either ambient occlusion or global illumination.

	Produces	Inputs				Output	Utilizes Off-screen Features	Underlying Technology
		Flat Color	Direct Light Map	Depth Map	Normal Map			
NNAO [12]	AO	-	-	✓	✓	AO Image	-	MLP
DeepAO [26]	AO	-	-	✓	✓	AO Image	-	U-Net
Fast Screen-Space Global Illumination [22]	GI	✓	✓	✓	✓	Final Image	-	CNN
DeepIllumination [15]	GI	✓	✓	✓	✓	Final Image	-	GAN
Surrounding-Aware Screen-Space Global Illumination (our approach)	GI	✓	✓	✓	✓	GI Map	✓	GAN

Table 5.5: Comparison of Our Approach with Other Existing Approaches

In the Table 5.5, AO, GI, MLP, CNN, GAN represents Ambient Occlusion, Global Illumination, Multi Layer Perceptron, Convolutional Neural Network, Generative Adversarial Network respectively.

In our system which utilizes a generative adversarial network, we attempt to produce global illumination for the viewport presentation. However, the output of our implementation network is not the final image that is showed on the screen, rather a globally illuminated lightmap that will be used in blending with the albedo/flat-color map before it gets presented on to the screen. The input it consumes is a feature-buffer set that contains four elements which are flat-color or albedo map, camera

local normal map, depth map, and combined lumination map. All of these feature buffers are rectangular 360° images (global views) of the surrounding of the camera, which are composed from six set of images captured from six different orientations.

The implementation of NNAO [12] generates ambient occlusion using a shallow neural network. Prediction of ambient occlusion do not require color information. The input to its network is screen-space feature buffers that only consists of depth maps and normal maps. It is only concerned with what is available on the screen and not whatever is off the screen. For ambient occlusion, it makes sense since it merely adds dark patches to corners and conjoined regions of 3D geometry already lit by ambient light, which adds some fancy effects to renders but does not bring realistic lighting. Offscreen light sources do not have any affect on the outputs.

DeepAO [26] also computes ambient occlusion by taking screen-space depth map and normal map into account. However, it utilizes a deep neural network instead of a shallow one. Even depending on a deep neural network, it still outperforms the previously mentioned NNAO in terms of output accuracy and runtime performance. The output ambient occlusion image is blended using a composition shader with a flat-color buffer before presenting on to the screen.

Fast Screen Space Global Illumination [22] attempts to solve the global illumination problem by employing a simple multi-layered convolutional neural network. The network consumes screen space feature buffers such as diffuse color, normal map, depth buffer/z-buffer, and local illumination map and tries to produce an image that is presentable to the screen. Contrary to our approach, this method generates the end image directly, rather than producing an intermediate lightmap containing global illumination that is further processable in the graphics pipeline. However, this approach works well for simple enough scenes.

Utilizing generative adversarial networks is performed by DeepIllumination [15]. Like most other methods, its inputs consists of depth maps, normal maps, albedo and direct lighting, and outputs a presentable image containing global illumination. This method does not produce intermediary global illumination lightmaps like our approach does as well. All inputs are screen-space in nature, so any objects or lights out of the screen do not have any effect on the final output. As a result, it suffers from off-screen inconsistencies.

Chapter 6

Discussion

In this chapter, we will be discussing about the limitations we faced in our thesis.

6.1 Limitations

Our implementation of generative adversarial network driven solution that utilizes beyond screen-space data by using surrounding-aware feature buffers is not a fool-proof solution to the global illumination problem. Surely, it can take into account off-screen lit surfaces or light sources and produce results accordingly, in contrast to existing SSGI solutions. But still, it has some drawbacks. Some of these include:

1. This implementation only specializes in diffuse global illumination. So, the specular indirect lighting will not work for this implementation. To overcome this, we might need to insert more channels into the network and extend our dataset and pipeline to include surface glossiness as input. Also, it is required to update the ground truth to include indirect lights caused by light bouncing on specular surfaces.
2. The system cannot work with indirect lights in terms of refraction.
3. If some self-emitting light source or lit surface is obstructed by some other object that is in front of the cameras, the network will not be able to take their effect into account due to getting less information about lighting, resulting in incorrect indirect lights. As shown in Figure [6.1], there is a narrow yet open space between the wall and ceiling through which lights can pass. However, it is not visible towards the camera due to being obscured, as a result, in the ray traced render the ceiling is lit, but not in the blended network generated result.



(a) Network Generated Result



(b) Ray traced image

Figure 6.1: Incorrect Illumination Due to Light Source Obstruction

Chapter 7

Conclusion and Future Plans

7.1 Conclusion

Employing screen-space features to perform various post-processing operations on top of rasterized renders can be viewed as a cheap alternative to expensive hardware ray-tracing to achieve photo-realistic results. It can still come with the expense of losing visual accuracy due to out-of-screen objects. Our implementation of deep learning based global illumination solution tries to improve existing screen-space global illumination techniques by viewing beyond the viewport, by being aware of the surroundings. Collecting enough graphical information about the environment rather than looking at a single direction brings visually consistent global-illumination while retaining speed.

7.2 Future Plans

Even though we have proposed a model, we are still planning on developing it and making it more sustainable and improved. We have also pointed more of our future goals. These are:

1. We will work with other aspect ratios rather than the 1:1 aspect ratio which we used here, making it more relevant with real screen resolutions such as 16:9 aspect ratio.
2. We will implement specular indirect lighting in our global illumination system.
3. We will generate a more diverse dataset and perform training and testing on it.
4. We will find a faster alternative to ray traced rendering
5. We will implement a system to run our GAN model in generic graphics API pipelines with the help of compute shaders.
6. We will reimplement our GAN models using Spatially-Adaptive Normalization [20] instead of Batch Normalization as an experiment. We are going to compare it with our existing approach.

References

- [1] A. Fujimoto, T. Tanaka, and K. Iwata, “Arts: Accelerated ray-tracing system,” *IEEE Computer Graphics and Applications*, vol. 6, no. 4, pp. 16–26, 1986.
- [2] Z. Wang, A. Bovik, H. Sheikh, and E. Simoncelli, “Image quality assessment: From error visibility to structural similarity,” *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 600–612, 2004. DOI: 10.1109/TIP.2003.819861.
- [3] T. Ritschel, T. Grosch, and H.-P. Seidel, “Approximating dynamic global illumination in image space,” in *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, ser. I3D ’09, Boston, Massachusetts: Association for Computing Machinery, 2009, pp. 75–82, ISBN: 9781605584294. DOI: 10.1145/1507149.1507161. [Online]. Available: <https://doi.org/10.1145/1507149.1507161>.
- [4] D. C. Barboza and E. W. G. Clua, “Gpu-based data structure for a parallel ray tracing illumination algorithm,” in *2011 Brazilian Symposium on Games and Digital Entertainment*, 2011, pp. 11–16. DOI: 10.1109/SBGAMES.2011.22.
- [5] C. Crassin, F. Neyret, M. Sainz, S. Green, and E. Eisemann, “Interactive indirect illumination using voxel cone tracing: A preview,” in *Symposium on Interactive 3D Graphics and Games*, ser. I3D ’11, San Francisco, California: Association for Computing Machinery, 2011, p. 207, ISBN: 9781450305655. DOI: 10.1145/1944745.1944787. [Online]. Available: <https://doi.org/10.1145/1944745.1944787>.
- [6] A. L. Maas, “Rectifier nonlinearities improve neural network acoustic models,” 2013.
- [7] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, *et al.*, *Generative adversarial networks*, 2014. arXiv: 1406.2661 [stat.ML].
- [8] G. Meneghel and M. Netto, “A comparison of global illumination methods using perceptual quality metrics,” Aug. 2015. DOI: 10.1109/SIBGRAPI.2015.52.
- [9] K. O’Shea and R. Nash, *An introduction to convolutional neural networks*, 2015. DOI: 10.48550/ARXIV.1511.08458. [Online]. Available: <https://arxiv.org/abs/1511.08458>.
- [10] O. Ronneberger, P. Fischer, and T. Brox, “U-net: Convolutional networks for biomedical image segmentation,” *CoRR*, vol. abs/1505.04597, 2015. arXiv: 1505.04597. [Online]. Available: <http://arxiv.org/abs/1505.04597>.
- [11] Scratchapixel, Aug. 2015. [Online]. Available: <https://www.scratchapixel.com/lessons/3d-basic-rendering/global-illumination-path-tracing>.

- [12] D. Holden, J. Saito, and T. Komura, “Neural network ambient occlusion,” in *SIGGRAPH ASIA 2016 Technical Briefs*, ser. SA ’16, Macau: Association for Computing Machinery, 2016, ISBN: 9781450345415. DOI: 10.1145/3005358.3005387. [Online]. Available: <https://doi.org/10.1145/3005358.3005387>.
- [13] P. Isola, J. Zhu, T. Zhou, and A. A. Efros, “Image-to-image translation with conditional adversarial networks,” *CoRR*, vol. abs/1611.07004, 2016. arXiv: 1611.07004. [Online]. Available: <http://arxiv.org/abs/1611.07004>.
- [14] T. Luiz, *Rasterization algorithms - computer graphics*, Sep. 2017. [Online]. Available: <https://medium.com/@thiagoluiz.nunes/rasterization-algorithms-computer-graphics-b9c3600a7587>.
- [15] M. M. Thomas and A. G. Forbes, *Deep illumination: Approximating dynamic global illumination with generative adversarial network*, 2017. DOI: 10.48550/ARXIV.1710.09834. [Online]. Available: <https://arxiv.org/abs/1710.09834>.
- [16] B. O. Community, *Blender - a 3d modelling and rendering package*, Blender Foundation, Stichting Blender Foundation, Amsterdam, 2018. [Online]. Available: <http://www.blender.org>.
- [17] R. Liu, J. Lehman, P. Molino, *et al.*, “An intriguing failing of convolutional neural networks and the coordconv solution,” *Advances in neural information processing systems*, vol. 31, 2018.
- [18] T.-C. Wang, M.-Y. Liu, J.-Y. Zhu, A. Tao, J. Kautz, and B. Catanzaro, “High-resolution image synthesis and semantic manipulation with conditional gans,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018.
- [19] Y. Wang, C. Wu, L. Herranz, J. van de Weijer, A. Gonzalez-Garcia, and B. Raducanu, *Transferring gans: Generating images from limited data*, 2018. DOI: 10.48550/ARXIV.1805.01677. [Online]. Available: <https://arxiv.org/abs/1805.01677>.
- [20] T. Park, M.-Y. Liu, T.-C. Wang, and J.-Y. Zhu, “Semantic image synthesis with spatially-adaptive normalization,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2019.
- [21] A. Paszke, S. Gross, F. Massa, *et al.*, “Pytorch: An imperative style, high-performance deep learning library,” in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché-Buc, E. Fox, and R. Garnett, Eds., Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: <http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf>.
- [22] B. Peng and P. Poulin, “Fast screen space global illumination,” 2019.
- [23] J. Thies, M. Zollhöfer, and M. Nießner, “Deferred neural rendering: Image synthesis using neural textures,” *ACM Trans. Graph.*, vol. 38, no. 4, Jul. 2019, ISSN: 0730-0301. DOI: 10.1145/3306346.3323035. [Online]. Available: <https://doi.org/10.1145/3306346.3323035>.
- [24] F. Dabak, M. Milk, M.-A. Zhong, N. Lunsingh Tonckens, J. Lähdemaa, and R. Velden, “Analysis of nvidia corporation,” Apr. 2020.

- [25] H. Xin, S. Zheng, K. Xu, and L.-Q. Yan, “Lightweight bilateral convolutional neural networks for interactive single-bounce diffuse indirect illumination,” *IEEE Transactions on Visualization and Computer Graphics*, 2020.
- [26] D. Zhang, C. Xian, G. Luo, Y. Xiong, and C. Han, “Deepao: Efficient screen space ambient occlusion generation via deep network,” *IEEE Access*, pp. 64 434–64 441, 2020. DOI: 10.1109/ACCESS.2020.2984771.
- [27] *Eevee is losing!* Feb. 2021. [Online]. Available: <https://www.youtube.com/watch?v=2Dh2Sf0kf70>.
- [28] O. S. Faragallah, H. El-Hoseny, W. El-Shafai, *et al.*, “A comprehensive survey analysis for present solutions of medical image fusion and future directions,” *IEEE Access*, vol. 9, pp. 11 358–11 371, 2021. DOI: 10.1109/ACCESS.2020.3048315.
- [29] Z. Hao, A. Mallya, S. Belongie, and M.-Y. Liu, *Ganecraft: Unsupervised 3d neural rendering of minecraft worlds*, 2021. DOI: 10.48550/ARXIV.2104.07659. [Online]. Available: <https://arxiv.org/abs/2104.07659>.
- [30] A. Kuznetsov, K. Mullia, Z. Xu, M. Hašan, and R. Ramamoorthi, “Neumip: Multi-resolution neural materials,” *ACM Trans. Graph.*, vol. 40, no. 4, Jul. 2021, ISSN: 0730-0301. DOI: 10.1145/3450626.3459795. [Online]. Available: <https://doi.org/10.1145/3450626.3459795>.
- [31] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An introduction to statistical learning: With applications in R*. Springer, 2022.
- [32] S. R. Richter, H. Abu Al Haija, and V. Koltun, “Enhancing photorealism enhancement,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pp. 1–1, 2022. DOI: 10.1109/TPAMI.2022.3166687.
- [33] *What is minecraft? build, discover realms and more*, Sep. 2022. [Online]. Available: <https://www.minecraft.net/en-us/about-minecraft>.
- [34] *Overview of gan structure*. [Online]. Available: https://developers.google.com/machine-learning/gan/gan_structure.
- [35] *Screen space global illumination*. [Online]. Available: <https://docs.unrealengine.com/4.27/en-US/BuildingWorlds/LightingAndShadows/ScreenSpaceGlobalIllumination/>.
- [36] *Screen space global illumination: High definition rp: 14.0.3*. [Online]. Available: <https://docs.unity3d.com/Packages/com.unity.render-pipelines.high-definition@14.0/manual/Override-Screen-Space-GI.html>.