Deep Illumination-Driven Light Probe Placement

Andreas Tarasidis

Diploma Thesis

Supervisor: Ioannis Fudos

Ioannina, July 2025

Τμήμα Μηχ. Η/Υ & Πληροφορικής

Πανεπιστήμιο Ιωαννίνων

Department of Computer Science & Engineering

University of Ioannina

Dedication

To my mother, father, and brother, who always helped me achieve my goals.

Acknowledgments

First and foremost, I would like to express my sincere gratitude to my supervisor, Prof, Ioannis Fudos, for his invaluable guidance, feedback, and support. As an advisor, he always made time, was extraordinarily patient, and his suggestions were instrumental in the completion of this thesis.

I am also grateful to the friends I've made over the years. Their camaraderie has made my academic journey truly worthwhile.

This journey wouldn't have been possible without them, so finally, but certainly not least, I extend my deepest gratitude to my family. Without their encouragement, boundless support, and unwavering belief in my potential, my works would never have reached fruition.

I dedicate this thesis to them.

Abstract

Realistic lighting is a cornerstone of visually compelling 3D graphics. Unity's Light-Probe system offers an efficient way to capture and interpolate baked Global Illumination (GI) data across dynamic objects in scenes. However, manual placement of light probes in complex scenes is both time-consuming and error-prone, greatly delaying the iteration process when making 3D applications. This thesis presents an automated, deep learning-based approach that predicts per-point importance scores for light probe placement using a PointNet-inspired neural network.

We first generate a regular 3D point grid that conforms to the user-defined arbitrarily-shaped bounds of the scene. We sample per-point lighting information, including spherical harmonics, light-, normal-, and RGB- variance, and occlusion factor as well. These features capture important information that drive GI accuracy. The data is then converted into a concise feature vector at each location, used to then train the PointNet-style AI model that consumes an arbitrary-length list of such feature vectors and outputs a probability in the range 0-1, depicting how vital it is to place a light probe at each point on the grid.

To deploy in Unity, the trained model is then exported to an .ONNX file and imported via Sentis, the official Unity package for handling AI models inside a Unity Runtime; at edit-time, it ingests per-point scene data and returns per-point importance values. Predicted high-importance locations are then used to populate a Unity LightProbeGroup object, giving developers immediate, visually appropriate probe distributions, with easy to control thresh-holding if higher- or lower-importance locations are desired.

We demonstrate that our AI model generalizes across grid sizes and shapes without retraining, as well as giving immediate results for any scene. Although our evaluation remains mostly qualitative, based on visual inspection of GI results and light-probe placement across a variety of indoor and outdoor scenes, we consistently observe that the generated probe layouts capture important scene light-data with minimal or no manual tweaking. By replacing manual probe placement with a simpler AI-based workflow, artists and developers save time and achieve a faster iteration process throughout the development of a 3D application.

Περίληψη

Ο αληθοφανής φωτισμός είναι ο ακρογωνιαίος λίθος των οπτικά ελκυστικών τρισδιάστατων γραφικών. Το σύστημα φωτο-ανιχνευτών (Light-Probe) της μηχανής γραφικών Unity παρέχει έναν αποδοτικό τρόπο καταγράφης και να παρεμβολής προπαρασκευασμένων δεδομένων παγκόσμιου φωτισμού (Global Illumination) προς όλα τα δυναμικά αντικείμενα μιας σκηνής. Πάραυτα, η χειροκίνητη τοποθέτηση των φωτο-ανιχνευτών σε πολύπλοκες σκηνές είναι μια χρονοβόρα διαδικασία, αλλά και επιρρεπής σε λάθη, δημιουργώντας μεγάλες καθυστερήσεις κατά την διάρκεια κατασκευής τρισδιάστατων εφαρμογών. Η διπλωματική αυτή παρουσιάζει μία αυτοματοποιημένη μέθοδο βαθιάς μάθησης η οποία προβλέπει τον βαθμό σημαντικότητας ανά σημείο για τα σημεία τοποθέτησης των φωτο-ανιχνευτών, χρησιμοποιώντας ένα νευρωνικό δίκτυο εμπνευσμένο από το PointNet.

Αρχικά δημιουργούμε ένα κανονικό τρισδιάστατο πλέγμα σημείων το οποίο προσαρμόζεται στα αυθαίρετα δομημένα όρια της σκηνής, ορισμένα απο τον χρήστη. Δειγματολειπτούμε πληροφορίες φωτισμού ανά σημείο, συμπεριλαμβάνοντας σφαιρικές αρμονικές, διακυμάνσεις φωτισμού, κανονικών επιφάνειας, και RGB, όπως και παράγωντα απόφραξης. Τα χαρακτηριστικά αυτά εμπεριέχουν σημαντικές πληροφορίες που ωθούν την ακρίβεια του παγκόσμιου φωτισμού. Στην συνέχεια τα δεδομένα δειγματοληψίας μετατρέπονται σε ένα συνεκτικό διάνυσμα χαρακτηριστικών ανά σημείο, και χρησιμοποιούνται για την εκπαίδευση του PointNet-τύπου μοντέλου τεχνητής νοημοσύνης. Το μοντέλο αυτό καταναλώνει μία αυθαίρετου μήκους λίστα από συνεκτικά διανύσματα χαρακτηριστικών και εξάγει μια πιθανότητα σε εύρος 0 έως 1, η οποία αναπαριστά την κρισιμότητα τοποθέτησης ενός φωτο-ανιχνευτή σε κάθε σημείο στο πλέγμα.

Στην μηχανή Unity, το εκπαιδευμένο μοντέλο εξάγεται σε ένα αρχείο τύπου `.ONNX` και εισάγεται μέσο του Sentis, του επίσημου πακέτου της Unity για χειρισμό μοντέλων τεχνητής νοημοσύνης εντός του εκτελέσιμου της Unity˙ κατα την επεξεργασία, καταναλώνει δεδομένα σκηνής ανά σημείο και επιστρέφει τιμές κρισιμότητας ανά σημείο. Οι προβλεπόμενες τοποθεσίες υψηλής σημασίας χρησιμοποιούνται για να συμπληρώσουν μια ομάδα φωτο-ανιχνευτών, αντικείμενο της Unity, παρέχοντας στους χρήστες άμεση και οπτικά κατάλληλη κατανομή των φωτο-ανιχνευτών, με ευκολόχρηστη κατωφλίωση όταν υψηλότερης ή χαμηλότερης σημασίας τοποθεσίας είναι επιθυμητές.

Επιδεικνύουμε ότι το τεχνητής νοημοσύνης μοντέλο μας γενικεύει ανάμεσα σε πλέγματα με διάφορα μεγέθη και σχήματα, χωρίς την ανάγκη επανεκπαίδευσης, όπως και ότι παρέχει άμεσα αποτελέσματα για κάθε σκηνή. Παρόλο που η αξιολόγησή μας παραμένει κυρίως ποιοτική, βασιζόμενη στον οπτικό έλεγχο του αποτελέσματος παγκόσμιου φωτισμού, όπως και τα σημεία τοποθέτησης των φωτο-ανιχνευτών σε ένα εύρος από εσωτερικών και εξωτερικών χώρων, παρατηρούμε συστηματικά πως η παραγόμενη διάταξη των φωτο-ανιχνευτών περιλαμβάνει σημαντικά δεδομένα φωτισμού της σκηνής με ελάχιστη ή μηδενική χειροκίνητη προσαρμογή. Αντικαθιστώντας την χειροκίνητη τοποθέτηση των προβες με την απλή χρήση ενός μοντέλου τεχνητής νοημοσύνης, οι καλλιτέχνες και οι προγραμματιστές κερδίζουν χρόνο και επιταχύνουν την διαδικασία ανάπτυξη μιας τρισδιάστατης εφαρμογής.

List of Figures

List of Algorithms

Table of Contents

Chapter 1

Introduction

Modern interactive 3D applications, like video games, VR/AR apps, simulators etc., depend on believable lighting interactions with the objects of a 3D scene to achieve the desired visual goals, while trying to maintain real-time frame-rate budgets, typically above 30 Frames per Second (FPS). Achieving visual fidelity and performance can be a difficult task and sometimes impossible with the given hardware specifications of the device. For that reason, modern real-time rendering engines, e.g. Unity, Unreal Engine, Godot and others, depend on a number of methods to balance those metrics.

The illumination of any scene can be split into two very simple categories. Direct Illumination, the light that travels unoccluded from a light source to a surface of an object, is typically handled with techniques like shadow-mapping or screen-space shadows, yielding crisp, high-framerate-capable shadows, but lack in inter-surface light transport situations. In contrast, Indirect Illumination, or Global Illumination (GI), captures light that has bounced or refracted off one or more surfaces, producing soft shadows, color bleeding, and contextually rich shading.

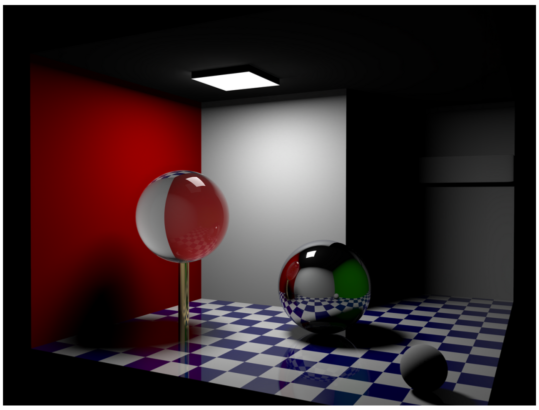

Figure 1.1: Scene lighting with direct illumination only. By Barahag - Own work, CC BY-SA 4.0
https://commons.wikimedia.org/w/index.php?curid=88541991

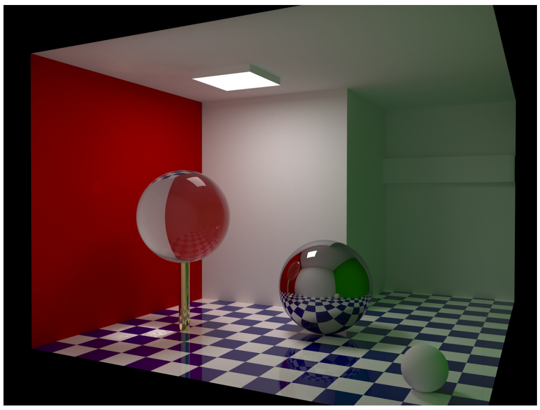

Figure 1.2: Scene lighting with Global illumination. By Barahag - Own work, CC BY-SA 4.0
https://commons.wikimedia.org/w/index.php?curid=88540888

The field-standard for accurate lighting and shadows in a scene is Path-Tracing, a method that tracks every light ray and any interactions it has with the objects of a 3D scene and calculates the resulting color for each pixel of the screen. Such approach remains prohibitively expensive for most interactive applications, so real-time systems employ pre-computation and approximation of the illumination of the scene; static geometry is baked into lightmaps that store per-texel irradiance, while dynamic elements sample from irradiance volumes or light probes, sparse 3D points whose spherical-harmonic coefficients are interpolated at runtime.

Screen-space GI methods typically approximate a limited number of light-ray bounces directly from the camera's depth buffer, but suffer from missing contextual information outside the camera's view frustum and temporal instability. Voxel-based approaches (e.g., cone-tracing through a low resolution 3D grid) enable more dynamic multi-bounce effects at the cost of memory, processing cost and potential blurring of fine detail.

Across all these techniques, the central challenge is allocating a strict millisecond-scale budget to indirect illumination while maintaining consistency across static and dynamic scene content, avoiding visible seams when blending baked and runtime solutions and fitting within GPU memory constraints.

Light probes, in particular, represent a compelling middle-ground, flexible enough to illuminate moving objects without rebaking yet compact enough for real-time evaluation, making their optimal placement a critical factor in any high-quality GI pipeline.

1.1 Related Work

There is an abundance of work in the literature addressing the problem of Global Illumination. These studies aim to achieve realistic lighting in 3D scenes by employing various approaches and techniques, each offering unique advantages and disadvantages, but they share a common goal: to maximize visual fidelity while minimizing computational costs.

1.1.1 Offline Methods

Offline Illumination methods refer to techniques that are not viable for real-time applications and are therefore used only in situations where the importance of high visual fidelity far outweighs the need for computational speed, typically in non-interactive 3D renders, most commonly in movies or pre-rendered scenes. Classic Path-Tracing, first introduced in 1986 (Kajiya 1986), tracks the movement of a photon ray emitted from a source, typically the camera, and simulates physics interactions to calculate the color of each screen pixel accurately. The immense computational cost of path-tracing led to the development of performance improvements, such as the Metropolis Light Transport (MLT) method introduced in 1997 (Veach and Guibas 1997), and variants like bi-directional Path-Trace (Lafortune and Willems 1993), which build on Monte-Carlo algorithms (Lafortune 1996).

1.1.2 Online Methods

In contrast, online methods aim to calculate GI interactions in real-time, most commonly used in interactive applications like video games or simulations. They try to balance performance and accuracy, a task that is often difficult due to the processing cost of the calculations for a realistic result. Therefore, these methods take shortcuts, either approximating the GI interactions to a certain degree to maintain framerate budgets, or by precomputing some of the data, wherever possible.

Traditional Methods

Techniques that precompute the illumination of a scene only do so for static geometry; objects in the scene that will never change their position, rotation or scale. The algorithms "bake" the required information onto texture maps, which are rendered as such when needed. Light-mapping is one such technique. It precomputes surface brightness and has a low runtime cost. The game Quake was the first interactive application that used lightmaps for rendering GI (Wikipedia contributors 2025).

Another early technique is the Irradiance Volumes algorithm (Greger et al. 1998), which scatters spherical-harmonic (SH) irradiance samples on a 3D grid on the scene. At runtime, lighting is interpolated from the nearest SH cells; this underlies many probe systems, like Unity's light-probe system that implicitly implements a sparse irradiance volume.

More recent static-GI algorithms include Light Field Probes (McGuire et al. 2017). Light Field Probes extend standard irradiance probes by additionally storing per-texel visibility for each probe. Furthermore, (Xu et al. 2022) introduce Discrete Visibility Fields for static ray-traced lighting. The method precomputes occlusion masks stored in a uniform voxel grid, and at runtime, rays that hit a cell use the stored precomputed masks to quickly cull visibility, skipping geometry already known to be occluded.

Unity's new Adaptive Probe Volumes (APV) build on irradiance volumes by automatically populating a grid, with density matched to local geometry. APV then performs per-pixel probe sampling; each pixel blends from the eight nearest probes (Unity 2025).

Additionally, there are methods that don't focus on Probes for GI. A prevalent example is Unreal Engine's Lumen, a dynamic GI and reflections system that uses a hybrid tracing approach; It starts with a cheap screen-space or signed-distance-field ray cast, and then falls back to more expensive methods like hardware ray tracing (EpicGames 2025).

NVIDIA has also developed RTXGI, a GPU-accelerated library implementing Dynamic Diffuse GI, using a volumetric grid of irradiance probes, which update every frame using hardware-accelerated ray tracing, creating accurate results at the cost of hardware-restricted algorithms and a relatively escalated cost of calculation (Nvidia 2024).

In 2011 (Crassin et al. 2011) , a Voxel Cone Tracing (VCT) technique was introduced to approximate real-time GI. In VCT, the scene's static geometry and lighting are "voxelized" into a 3D texture with multiple levels of mipmapping, containing radiance and opacity. At runtime, indirect illumination is approximated by tracing a few low-resolution "cones" from each surface sample into the aforementioned voxel grid, summing the values from regions of voxels.

Even though there are numerous methods trying to solve real-time GI issues, a big percentage of them tend to revolve around probes of various types; most commonly calculating irradiance values among other high-importance metrics. Therefore, it is vital for a 3D scene to have proper probe placement for best results. There are a few methods that try to automate that process, often by placing the probes in a regular grid and only removing the probes that are inside objects, but that can lead to over-sampling, leading to performance costs, mainly in memory usage budgets. Furthermore, some techniques try to remove additional probes using heuristic methods, therefore approaching optimal placement, but with a significant precomputational cost.

In (Wang et al. 2019), an automatic non-uniform placing scheme is introduced, which uses 3D scene skeletons and gradient-descent refinement to cover important locations without redundant probes. A very recent work formulates geometry-based optimization of probe placement using various mesh features, to further improve the lighting in VR/AR scenarios (Teuber et al. 2024).

Similarly, (Vardis, Vasilakis, and Papaioannou 2021b) approach the problem by starting with a probe set on a dense grid and iteratively removing the least-important probes using radiance error tests, preserving the global light field while minimizing probe count.

AI-based Methods

Recently, AI-assisted methods have started to be developed in order to improve GI in 3D applications, specifically in probe-based solutions. In (Guo et al. 2022), they propose a hybrid neural probe GI. They use a gradient-based search to re-project stored probe radiance for any view, therefore eliminating parallax, and then apply a small neural network to reconstruct high-quality images from low-resolution probe data. Related, a Neural Light Field Probes method has been introduced (You, Geiger, and Chen 2024), which works by decomposing a scene into a grid of trainable light field probes. Each of these probes encodes local radiance and visibility in a compact feature map. Finally, a neural network optimizes these probes so that the summation of their contributions reproduces the full scene lighting.

1.2 Thesis Structure

The structure of the remainder of this thesis will be described shortly. Chapter 2, titled Background, covers important information about light probes and their implementation, describes the AI model basis that was used for our implementation, and introduces the tools and technologies that were used in this thesis. Chapter 3, titled Our Approach, presents our method, describes the implementation of the algorithms used, and explains how each part is combined to create the Light-Probe Neural Network (LPNN), our neural network system that attempts to speed up light probe placement in Unity 3D scenes by predicting importance values for the given grid set and placing only the most vital light probes, affected by a user-controlled threshold value. Each grid position gathers samples of a few metrics, which are then used by the neural network to decide whether or not it is vital to place a light probe in each individual cell of the 3D grid. A step-by-step process of creating the grid, getting the features out of the grid cells, and placing the probes and baking the global illumination inside Unity. Additionally, the feature set can be used to retrain the AI model, we explain how to create the labels needed for the process, and how to import the new model to Unity using Sentis for usage. Chapter 4, titled Experiments, presents an experimental comparative and qualitative evaluation between the proposed method and some of the already introduced algorithms. Finally, Chapter 5, titled Conclusions and Future Works, concludes the thesis and proposes directions for future work based on this thesis.

Chapter 2

Background

In this chapter we introduce some basic, required background for this thesis. First, we introduce light probes and the mathematical equations that define them. Then, we present the AI architecture that was the basis of our AI model. Finally, the tools and technologies used for this thesis are presented.

2.1 Light Probes

As mentioned previously, the idea of using discrete probes to capture scene lighting data traces back to early GI research. In the paper (Greger et al. 1998) introduced the irradiance volume, a 3D grid of sample points storing the irradiance field to approximate GI in complex scenes. A light probe samples the incident radiance at a point in empty space from all directions. Often just the diffuse component of the radiance is captured, since it most commonly varies smoothly, so it can be compactly represented by projecting the lighting onto a truncated spherical harmonic (SH) basis. Third-order SH is most commonly used, storing 9 coefficients per color channel, abbreviated to L2-SH.

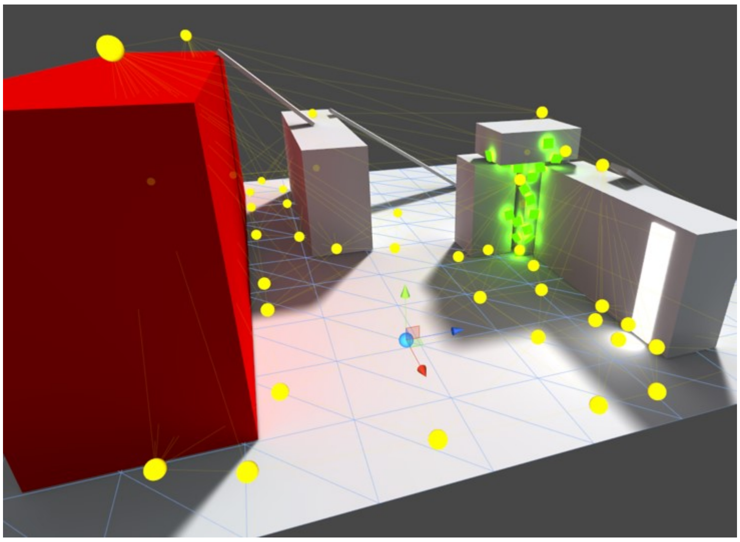

Figure 2.1: A 3D Scene showing a few light probes placed in important locations (Unity 2016).

The light probe layout is then used to create tetrahedral shapes between the light probes. Any dynamic object that is within one of those shapes interpolates lighting information for the four light probes comprising the corners of the shape, to illuminate the surface of the dynamic object as accurately to the ground-truth baked illumination as possible.

2.2 Spherical Harmonics

Spherical Harmonics (SH), first introduced by Pierre Simon de Laplace, are a method of storing information on a point in space. They are categorized in structures called orders. We are interested in third-order SH, since they represent a good middle ground between storage size, computational cost and accuracy. SH are often described as the Fourier Series of functions on the surface of a sphere, breaking down any pattern of light on a sphere into a set of basis frequencies. The order of SH depicts the amount of data we capture, third-order SH, noted as L2 SH, store the first three bands of data, resulting in 9 coefficients per color channel. Bands represent the individual frequencies; the Zeroth band captures the overall average lighting present in that position in space, the First band captures simple directional gradients, and the Second band captures quadratic variations, e.g. gentle light gradients and their shadows.

2.3 PointNet

PointNet (Qi et al. 2017) is a neural-network architecture designed to work directly on unordered 3D point-clouds; a collection of points without any required grid connectivity. Each point passes through a small MultiLayer Perceptron (MLP), extracting a feature vector that describes its local attributes, e.g. color and normal. After per-point features are computed, PointNet aggregates them into a global descriptor by applying a symmetric operation, usually max-pooling across all points, capturing the strongest signal from the features. Then, the global descriptor is concatenated back to the per-point features, resulting in every point having knowledge about both its characteristics and the broader context. Finally, a per-point MLP refines these combined values into task-specific outputs, commonly classification scores or per-point importance metrics.

Since there is no fixed grid, meaning the points are assumed to be unordered and irregular, traditional CNNs can't be applied directly. Additionally, since PointNet predicts values per-point, it can be used to handle point clouds of any shape and point amount without retraining, limited only by the system's memory. This makes PointNet a fitting candidate to base our model on, with the implementation being the only varying factor.

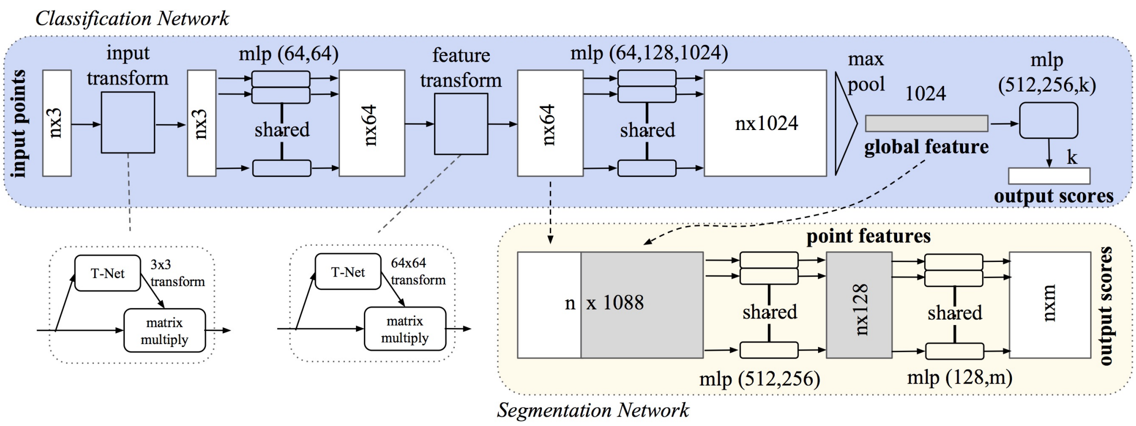

Figure 2.2: PointNet architecture. Image from Qi et al. 2017.

2.4 Tools

In this section, technologies, tools, and assets used throughout this thesis will be briefly presented.

2.4.1 Graphics Engine

For the implementation of this thesis, a stable and well-documented engine with a variety of capabilities, especially supporting light probe features, was necessary. For that reason, we decided to use the Unity Engine. Unity is used globally for computer applications, ranging from 2D and 3D interactive applications like video games, simulations of physics interactions like liquid simulations, and cinematic films with realistic or stylized visuals. Unity additionally supports light probes internally, under the name Light Probe Groups, an object that contains all individual light probes of a scene and handles the interpolation and mapping of dynamic objects passing through the scene that are affected by GI.

For programming in Unity, the programming language C# is used, as it is internal to the engine. Unity's backend codebase is made using C#, making any custom scripts made in the same language trivial to connect to the system, using the provided methods. Additionally, Unity exposes a number of methods and libraries for easily accessing information and variables of the current scene, and allows for editing of a percentage of those variables on edit- and run-time. Furthermore, exposing custom fields and variables is trivial, allowing for easier handling of assets via the editor window without requiring algorithms that control data on the hard disk or memory.

2.4.2 Sentis

For the purposes of this thesis, a tool that allows us to run AI models inside Unity was needed. For that purpose, we used Sentis, a neural-network inference library for Unity. It is able to detect and import pre-trained AI models into Unity as assets, run them both in runtime and edit time, and utilize the end-user's device compute, CPU or GPU. Sentis is able to capture an .onnx file containing the AI model, and then exposes a variety of functions via programming code to allow the developer to send data to the model and capture the output of the model.

2.4.3 3D Scenes

For training and testing the AI model, we used the Sponza scene (McGuire 2017), the Office scene (CG AUEB 2021), and the Corridor scene (Vardis, Vasilakis, and Papaioannou 2021a). The scenes underwent some manual editing to make them better suit our needs, remove assets deemed unnecessary for our goals, or to import them properly inside Unity along with their textures and required objects. These edits are minimal and not vital to the thesis. Therefore, they will not be explicitly described. The Corridor scene contains areas with illuminated and shadowed areas in close proximity, as well as differently colored lights, providing a gradual change in the lighting information. The office scene is also an indoor scene, but with a bigger emphasis on a higher number of different light sources in close proximity to each other, creating areas of high significance. Finally, the Sponza scene is an outdoor environment illuminated only by a light source depicting the sun. Any additional lighting is created by indirect illumination, meaning light that has bounced off of the different static objects in the scene.

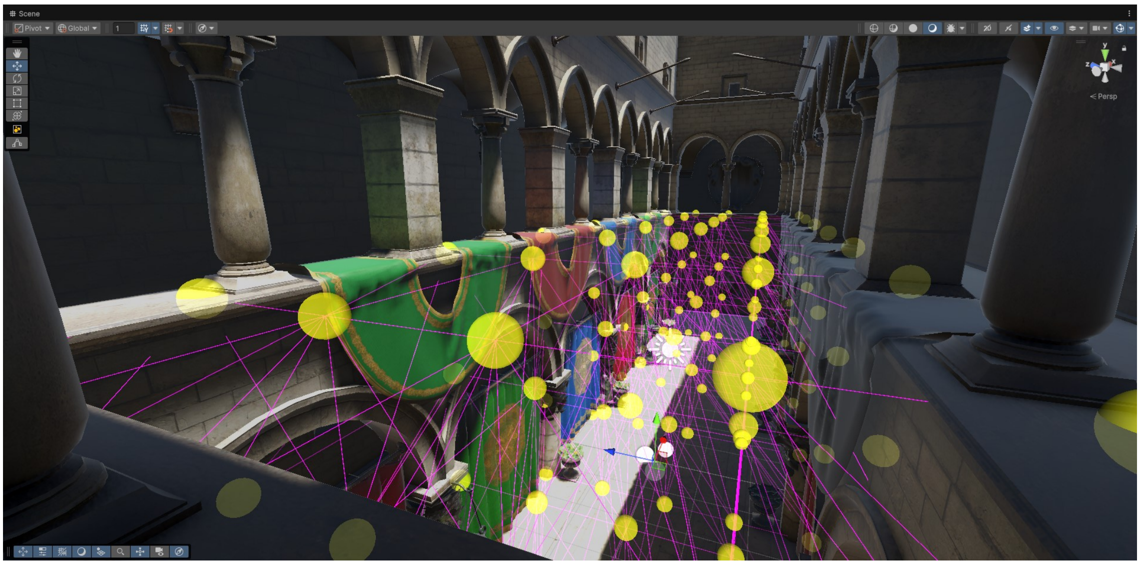

Figure 2.3: The Sponza Unity Scene showing light probes placed on a grid (McGuire 2017).

2.4.4 Python and TensorFlow

For this thesis, Python was used as the language of choice for the creation of our Deep Learning AI Model. Python is an interpreted programming language. The high-level nature of the language makes it ideal for general purpose scenarios and easy to work with. Python was used for its simplicity, making the reading, preparation, post-processing, and saving of the data a fast process during the development of LPNN.

TensorFlow is a free and open-source software library used for machine-learning and artificial-intelligence applications. It supports a variety of well-known programming languages, but for this thesis we used the Python version for creating and training our model. The choice of Python together with TensorFlow as the language and library for this task allows us to focus entirely on the implementation of the tool, and save time configuring the language and the requirements.

Chapter 3

Our Approach

In this chapter, we will present the pipeline of the tool and its implementation. The chapter will be split into five parts, describing the process of collecting the features, how labels are created, LPNN training in python, and predicting light probe positions using the trained model accordingly. Lastly, we will present the tool inside the Unity Editor.

3.1 Feature Collection

A necessary step before collecting scene and lighting features is to first place points in the scene to collect data per-point. We decided to create an algorithm that places those points, called Evaluation Points (EP) henceforth, inside a collection of user-defined scene bounds of arbitrary dimensions. As shown in algorithm 1, we begin by placing the Evaluation Points on a volume that surrounds the collection of bounds the user defined, then we remove the points that are not within the bounds, as seen in figure 3.1.

Algorithm 1 Placement of Evaluation Points on a grid-like layout

Require: $cellsize > 0$

$EP = \emptyset$
for all $points \in volume$ **do**
  if $point \in bounds$ **then**
    $EP \leftarrow EP + point$ ▷ Append point to the set
  end if
end for
return EP

In algorithm 1, *cellsize* is the distance between each EP, and a value of 0 or negative values do not apply, therefore it is vital to require a *cellsize* bigger than 0. *Cellsize* is not directly shown inside the algorithm, but for the C# implementation it is necessary and the value is used often, therefore we show the requirement here. *Bounds* is the collection of 3D areas defined by the user, and *volume* are the bounds of that collection. As seen in figure 3.1, the rectangles shown in pink are three distinct areas defined by the user, defining where they want EP placed. Seen in red is the surrounding volumes of those bounds. We clearly see the EP, shown as yellow dots. They are present only inside the areas of the user defined bounds, but in a grid layout, regardless of the position of those bounds. The EP intentionally "overshoot" the bounds for completeness when calculating the feature data for each point, making sure we cover the entirety of the bounds given by the user.

After placing the EP, we use each point to calculate the data needed for the feature vectors. The aim of the LPNN model is to give an importance value to each of those EP

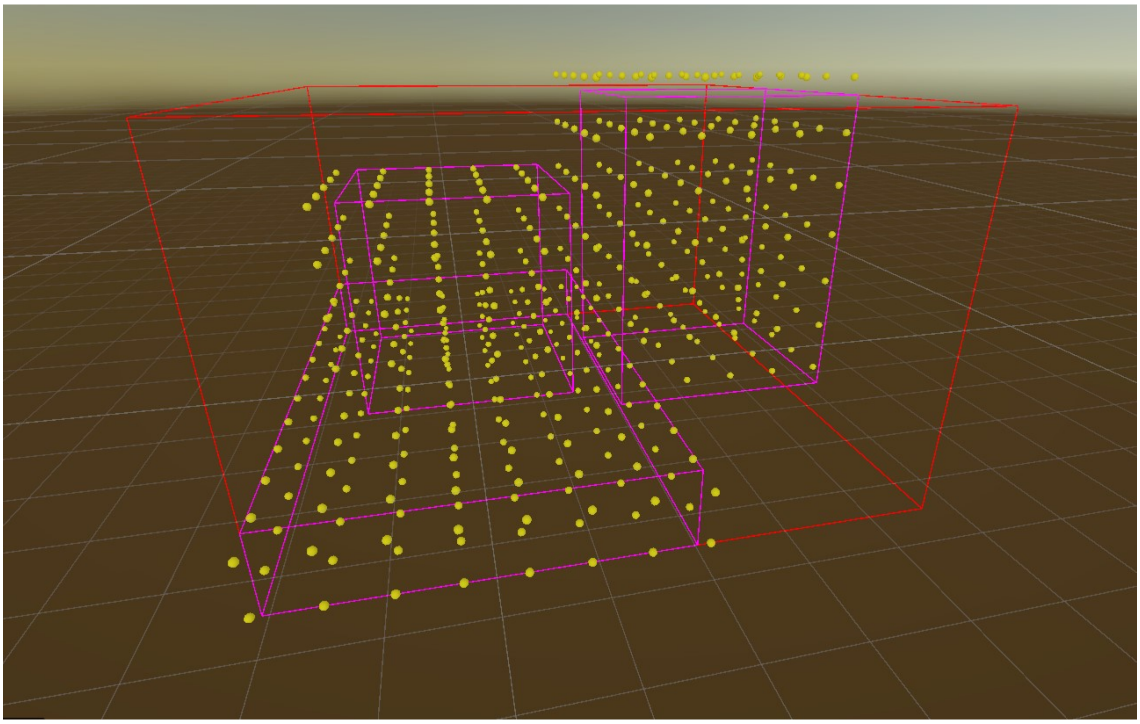

Figure 3.1: A 3D Scene showing the placement of Evaluation Points (shown in yellow) inside the user defined bounds (shown in pink), contained within the calculated volume (shown in red).

that it is given. Therefore, it is vital that it understands what defines a significant point via the features that it is trained on. Our knowledge dictates that an important location for a light probe is one that has great variance in illumination. Therefore, we include light-, RGB- and normal-variance as features for each point. Light-Variance dictates how much the light intensity changes around a point. RGB-Variance is similar, since it shows the change in Red, Green, and Blue light around each point, each channel being independent from the others. Normal-Variance is a value that makes locations like corners distinct from empty areas or from locations near a wall. Similarly, we also calculate Occlusion-Factor, a feature that gives each point a value depending on how empty or cluttered the area around it is. Normal-Variance together with Occlusion-Factor inform the model of areas that are enclosed or close to complex geometry. This makes the model able to distinguish these locations, even when all other values are identical, making it able to give different importance values to those distinct locations.

Along with the features mentioned above, we additionally include a vector that contains the spherical-harmonic values for a small amount of distinct directions around each point. The implementations of each feature collection method are shown in 2, 3, 4, 5, and 6.

Algorithm 2 Feature Extraction: Spherical Harmonics around a point

Require: $EP \neq \emptyset$
$values = \emptyset$
for all $points \in EP$ **do**
  $value \leftarrow evaluateSH(point)$
  $values \leftarrow values + value$ ▷ Append SH to the list
end for
return $values$

Algorithm 3 Feature Extraction: Light Variance around a point

Require: $EP \neq \emptyset$
Require: $samples \geq 1$ ▷ The amount of directions
1: $radius \leftarrow 1$ ▷ How far around the point to check
2: **for all** $points \in EP$ **do**
3: $luminances = \emptyset$
4: **for** $n = 1, \ldots, samples$ **do**
5: $direction \leftarrow$ random point on sphere with $radius$
6: $sh \leftarrow evaluateSH(direction)$
7: $luminance \leftarrow sh.R * 0.2126 + sh.G * 0.7152 + sh.B * 0.0722$ ▷ From RGB to luminance value
8: $luminances \leftarrow luminances + luminance$ ▷ Add luminance to the list
9: **end for**
10: $mean \leftarrow mean(luminances)$ ▷ Get the mean value of the luminances
11: $variance \leftarrow (luminances - mean)^2 \div samples$ ▷ Get the total variance of luminances
12: **end for**
13: **return** $variance$ per point

In line 3.7, to convert from RGB channel values to perceived light intensity of the pixel, also known as luminance, we use the formula shown in ITU-R 2015, under section 3 item 3.2. This formula is used to calculate how bright a pixel in regards to the RGB channel values it currently shows.

Similarly, we collect the RGB-Variance independently for each channel. The formula is similar to Light-Variance, with the exception of line 3.7, since it is not applicable. We need each channel RGB value to be unaffected by any conversion. This is seen in algorithm 4.

Algorithm 4 Feature Extraction: RGB Variance around a point

Require: $EP \neq \emptyset$
Require: $samples \geq 1$ ▷ The amount of directions
$radius \leftarrow 10$ ▷ How far around the point to check
for all $points \in EP$ **do**
$count \leftarrow 0$
for $n = 1, \ldots, samples$ **do**
$direction \leftarrow$ random point on sphere with $radius$
$sh \leftarrow evaluateSH(direction)$
end for
$meanRGB \leftarrow mean(sh.RGB)$
$RGBvariance \leftarrow (sh.RGB - meanRGB)^2 \div samples$
end for
return $RGBvariance.R, RGBvariance.G, RGBvariance.B$ per point

In line 5.12, we want the higher values to define the areas with more complex geometry. The dot product of vectors is a value between 0 and 1, therefore we subtract the dot product from 1, to invert the range of the variance.

Similarly to algorithm 5, algorithm 6 casts rays around each point, the amount determined by the *samples* variable, for a certain distance, determined by the *radius* variable. Each time a ray collides with any object, we increment a variable. At the end, we calculate the percentage of rays that collided to the total rays cast, and we return the result for each

Algorithm 5 Feature Extraction: Normal Variance around a point

Require: $EP \neq \emptyset$
Require: $samples \geq 1$ ▷ The amount of directions
$radius \leftarrow 1$ ▷ How far around the point to check
for all $points \in EP$ **do**
  $normals = \emptyset$
  for $n = 1, \ldots, samples$ **do**
    $direction \leftarrow$ random point on sphere with $radius$
    $hit \leftarrow castRay(direction)$
    if $hit == True$ **then**
      $normals \leftarrow normals + hit.normal$
    end if
  end for
  $mean \leftarrow mean(normals)$ ▷ Get the mean direction of the normals
  $variance \leftarrow (1 - dot(normals, mean)) \div samples$ ▷ Get the total variance of normals
end for
return $variance$ per point

Algorithm 6 Feature Extraction: Occlusion Factor around a point

Require: $EP \neq \emptyset$
Require: $samples \geq 1$ ▷ The amount of directions
$radius \leftarrow 10$ ▷ How far around the point to check
for all $points \in EP$ **do**
  $count \leftarrow 0$
  for $n = 1, \ldots, samples$ **do**
    $direction \leftarrow$ random point on sphere with $radius$
    $hit \leftarrow castRay(direction)$
    if $hit == True$ **then**
      $count \leftarrow count + 1$
    end if
  end for
  $factor \leftarrow count \div samples$ ▷ Percentage of hits versus misses.
end for
return $factor$ per point

point. Higher values describe points that are surrounded to geometry in close proximity.

After collecting all the features, we compact them into a feature vector per-EP, and save them into a file for use during training, or directly for light probe placement, as we will see shortly. As shown in section 3.5, it is also possible to collect feature vectors from multiple scenes at different times, as well as labels, to make the model more accurate by providing more input data.

3.2 Label Collection

The supervised-learning approach we decided on for this thesis requires that the model also receives labels as an input. Labels are values for each of the input feature vectors that describe the correct output for that feature vector. It is therefore vital to collect labels for each EP that we have also collected features for. For this purpose, we use LumiProbes (Vardis, Vasilakis, and Papaioannou 2021b) to collect labels by returning a True or False value, depending on if the cell of each Evaluation Point contains a probe placed by LumiProbes. The use of LumiProbes as the ground-truth was arbitrary. The system is constructed in a way that allows any method of ground-truth light probe placement to be used for label extraction, even manual placement. In algorithm 7 we check every EP with every light probe in the ground-truth list and set the label to True if there exists a light probe around a *cellsize* radius of that Evaluation Point.

Algorithm 7 Label Extraction per-EP

Require: $EP \neq \emptyset$
Require: $cellsize > 0$

for all $points \in EP$ **do**
    for all $probes \in groundTruth$ **do**
        if $probe \in cellsize$ around $point$ **then** ▷ If the probe is within the cell of each point...
            $label \leftarrow True$
        else
            $label \leftarrow False$
        end if
    end for
end for
return $labels$ per point

The nature of this label extraction method beckons for the ground-truth light probe placement to be close to the optimal placement, determined by the user's intuition, experience, and demands for the specific scene. We define a high-quality light probe layout in sub-section 4.2.1.

After collecting the labels into a list, they are saved into a file on the hard disk for use in training of the model, as seen in section 3.3. As mentioned previously, it is possible to collect labels from multiple scenes, creating a bigger dataset for the model to train on. The layout of the feature vectors stored in the files is as follows:

1. **L2 SH**, 6 RGBA values depicting the red, green, blue, and alpha channel in each of the axis in positive and negative directions. This results in 24 floating-point values.

2. **Light Variance**, the variance of irradiance strength as desribed beforehand.

3. **Normal Variance**, the variance of the normal vectors of the surrounding objects, if any.

4. **Occlusion Factor**, the percentage of rays that collided with an opaque object within a set distance to the total amount of rays casted.

5. **RGB Variance**, similar to Light Variance, depicts the change of color within a certain radius around the point. Each channel is calculated independently, resulting in 3 floating point values.

At the start of the file, a number describing the amount of features within each feature vector is prepended, for easier extracting during the pre-processing for the training step, seen in section 3.3. The format present is arbitrary, and it can be changed with no impact to the tool or the training of the model.

3.3 Model Training

The model basis for LPNN, as described in section 2.3, was a PointNet architecture (Qi et al. 2017). We followed the style loosely, mainly focusing on making a model that is able to predict importance scores per-point, and able to handle an arbitrary amount of input points without retraining.

3.3.1 Data Preparation

The feature vectors that were created from the previous steps are read from the files stored on the hard disk. For the labels, we convert each True or False line into numerical 1 or 0, respectively. Then, a Numpy array is created, containing all the values in order of appearance. For the feature vectors, a similar method is used. Additional logic is needed since there are multiple values per line, but the features are finally converted into an array of arrays. Each sub-array contains the values per-point, in order of appearance.

3.3.2 Model Architecture & Training

The architecture is as follows; First, three Convolution1D layers followed by three Batch Normalization layers expand the input from a feature vector of length 30 to 256. Each of these 6 layers are used one after the other, first a Convolution1D with dimensions 64x30x1x1 and a ReLu activation, followed by a Batch Normalization layer, then the next Convolution1D layer with dimensions 128x64x1x1 with the same activation function, etc. Then, in order for the model to have some knowledge of the global context, we use a Global Max Pooling 1D layer and a Global Average Pooling Layer 1D, concatenated together with each per point local context, resulting in a 512 dimensional feature layer.

In order to avoid averaging all neurons together for each point, we tile the global context, using a custom tiling function, making each neuron have knowledge of only the surrounding points, instead of every point in the scene. This ensures that each point's significance will only be affected by the surrounding lighting data of itself and the nearby points. However, this method also ensures that each point still has some knowledge of information beyond the tile that it was trained on, since the global pooling is done before the tiling. Therefore, each tile is only a piece of the total, but contains the information that was affected by the global pooling layers. This makes each point not completely agnostic to the global environment beyond the tile it was trained on, but it also makes the point more sensitive to surrounding data within each tile.

Finally, another Convolution1D and Dropout sequence is used to reduce the high-dimensional hidden layers into a single importance score $i \in [0, 1]$. The Dropout layers are important to reduce learning error rates by randomly deactivating a small set of neurons and their connections during the training process. The probability for drop-out is set to 0.3, meaning 30%. This sequence of layers are then directed into another Convolution1D layer with Sigmoid activation function, and trained for 50 epochs.The model is then compiled and saved in an .onnx file for import back into the Unity engine.

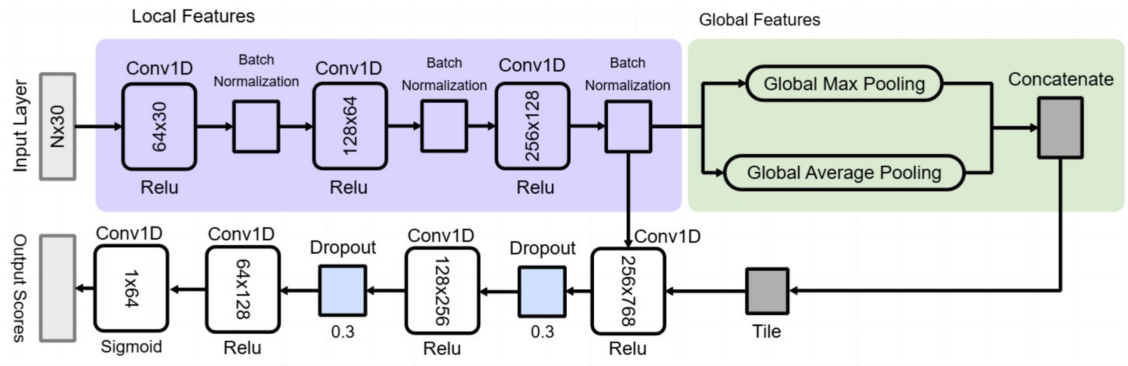

Figure 3.2: Neural Network architecture of our LPNN model.

The model was trained using 3485 total feature vectors, collected from a number of scenes, with multiple LPNN layouts and LumiProbes settings for each scene, ensuring the model has access to a variety of different scenarios to better understand the task, improving accuracy.

The architecture of the model, combined with the fact that the features do not contain any information about the scene itself, e.g. where lights, walls, windows etc. are placed, ensures that the model is fully agnostic to each scene, even if it was used for training. Different layouts for each scene can be used to increase the dataset, while avoiding overfitting, making the feature collection process easier.

3.4 Light Probe Prediction

In order to use the model to infer importance scores for the EP, we need to import it to Unity. For this purpose, we use Sentis. After model import, it can be used for our task. First, since the model is agnostic to each scene and each EP layout, we need to provide it with the feature vector list, containing the feature vector for each EP. This process is the same as in 3.1, therefore it is not described again. The only exception is that the list is not saved into a file at the end of the process, since it will be used directly inside Unity, therefore we only need the variable that contains the list object inside the code implementation.

First, it is necessary to convert the feature vectors into the same data format that was used when the model was trained using Python and TensorFlow. Providing any other format will result in the program not executing correctly, causing an error message or even a crash. This is because Sentis is able to execute model inference workloads on the GPU for parallelism and therefore faster execution. This is optional, since Sentis has a fallback feature. If no capable GPU is detected on the system, it uses the system's CPU instead. We have set Sentis to prefer using GPU compute for faster execution even with larger datasets, but the CPU fallback makes sure no additional logic is needed in cases where the system used does not contain a GPU.

After converting the vectors into the correct format, the compute device that was chosen is then sent the data and requested to infer using the model that we trained. This is done asynchronously, meaning that the execution of the code is not halted while the device is processing. It is therefore possible to request the inferred values back at a later time, but that is not necessary for our use-case. We save the inferred values to a file on the hard disk, meaning we can perform calculations on the values without affecting the original list, as we will see shortly, and remain able to read the original values without needing to re-infer using the model.

After inferring, the Light probe group can be populated according to the inferred importance scores. For that purpose, as seen in algorithm 8, it is necessary to pre-process all the values for better handling. It is not unlikely that the scores given by the model are very concentrated to some value, sometimes clusters forming in the data, making the use of the threshold system delicate. To fix this, we remap the range of the values from the minimum and maximum value to 0 and 1. This results in the values covering the entirety of the output range 0-1, instead of potentially only parts of it, spreading out any clustering of values.

Algorithm 8 Inferred Scores Range Remapping

Require: $inferredValues \neq \emptyset$ ▷ The inferred value list from the model.
1: $min \leftarrow$ inf ▷ Positive Infinity
2: $max \leftarrow -$inf ▷ Negative Infinity
3: **for all** $value \in inferredValues$ **do**
4:     **if** $value < min$ **then**
5:         $min \leftarrow value$
6:     **else if** $value > max$ **then**
7:         $max \leftarrow value$
8:     **end if**
9: **end for**
10: **for all** $value \in inferredValues$ **do**
11:     $value \leftarrow (value - min)/(max - min)$ ▷ After calculating minimum and maximum, we remap the values
12: **end for**
13: **return** $values$ list

The complete function for remapping a value x from a $[min1, max1]$ range to another $[min2, max2]$ range is as follows:

$$result = min2 + (max2 - min2) * ((x - min1)/(max1 - min1))$$

For our purposes, the ranges are $[min, max]$ and $[0, 1]$. Therefore, the function can be simplified to the one seen in 8.11.

With the values remapped, the threshold system can be used without delicate control. We decided on a percentile threshold system, meaning that the user can decide what percentage of the total amount of light probes they want to be placed, with the most important ones appearing first. The algorithm for this is simple, as seen in 9. First, we duplicate the list, leaving the original unaffected, which the duplicate is then sorted in a descending order. Then, from the ordered list, the top percentage of the values is kept, depending on the threshold value that is currently set by the user.

In line 9.1, we invert the threshold, since, as mentioned before, we want the higher-importance values to appear first. This means that with a 90% threshold, the top 10% light probes with the highest importance score should be placed, while the remaining 90% with lower significance should be ignored.

Finally, we set the positions of the Light Probe Group Unity object to be the ones that are within the threshold that we calculated, seen as the *positions* variable in the algorithm mentioned. As we will see in section 3.5, the light probes are now placed in the

Algorithm 9 Thresholded Placement of Light Probes

Require: $inferredValues \neq \emptyset$
Require: $threshold \in [0, 1]$ ▷ The threshold given by the user
$percentage \leftarrow ((1 - threshold) * inferredValues.count)$ ▷ Calculate the amount requested by the threshold value, from the total
$duplicate \leftarrow inferredValues.copy$
$duplicate \leftarrow sort(duplicate).copyAmount(percentage)$ ▷ Sort the duplicate list and keep only the desired amount
for all $value \in duplicate$ **do**
    $positions \leftarrow positions + inferredValues.indexOf(value)$ ▷ append the EP to the desired positions list
end for
$LightProbeGroup.positions = positions$
return $LightProbeGroup$

scene and visible to the user, completing the execution of the tool.

3.5 LPNN inside Unity Editor

In this section, we will describe the workflow of the tool and how a developer can speed up the process using LPNN. Additionally, the steps needed for data extraction are described, allowing for retraining of the LPNN model by the user.

3.5.1 Normal Execution of the Tool

For the tool to be executable inside the Unity Engine editor view, we implemented a GUI using Unity's *CustomEditor* attribute, allowing us to configure the layout via code for the Inspector view. First, as seen in figure 3.3, the user imports the tool into the scene by the same method when importing any built-in asset available.

An *LPNNParent* object is then created inside the scene's hierarchy, appearing in the related window. As we will see shortly, this object houses all the objects needed for execution of the tool. Namely, a *BoundingVolumes* object is added automatically, which will contain all the bounding volumes the user wants to be included when creating the EP grid. Additionally, a *VoxelsParent* object is created automatically when placing the EP using the necessary button.

In the case where the user wants a standard bounding box to be used, this can be specified using a simple box as a bounding volume. For more complex bounding volumes, more shapes can be added and the tool will show the new bounding areas, after recalculation. The complete bounding volume dimensions are shown in the editor window using fields that describe the Center point and the Extend of the bounding volume in each axis, as seen in 3.4. Setting up the area that the tool should place the EP in is a necessary step. Without it, error log messages will appear in the Unity Console when the user attempts to run almost any other step in the process. The *Calculate Bounds* can be clicked when the user has defined their bounding volumes. The system will execute a number of calculations to create the necessary objects for the next steps.

The *Visualise Bounds* flag is available to the user optionally, set to True as a default. In the True state, it reveals to the user the total volume, each bound, and the EP points, if set, inside the Scene window of the Unity editor, as seen in figure 3.1. Additionally, it is updated in real-time when the user changes the size of the *Voxel Size* that will be described shortly, making the process of finding the desired grid layout faster. If the user

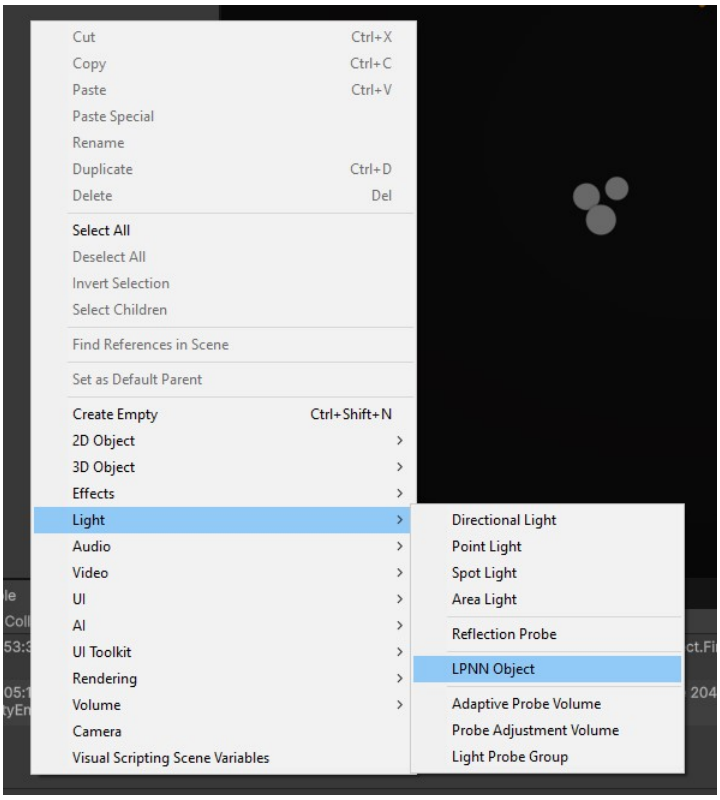

Figure 3.3: Figure showing the tool asset, called *LPNNObject*, available as a Menu Item under *Light* category, allowing for easy importing to any Unity scene.

disables the flag, none of the bounds or the EP will be shown, allowing for a visually simpler Scene view.

In order for the EP grid to be created, the *Place Evaluation Points* button is available. When clicked, the implementation of the algorithm 1 is executed, placing the EP on a grid inside the user-defined bounds, taking into consideration the *Voxel Size* field to determine the size of each cell.

For the EP to capture Global Illumination data properly, Unity must have created the necessary objects, maps, and assets for each point to collect the features from. However, this process is not always executed beforehand. for this reason, we added the *Bake Global Illumination* button, which requests a total recalculation of the scene's GI information from Unity, ensuring all data is available for the tool.

After baking the GI, the *Get Features* button can be clicked, which executes all the algorithms mentioned in section 3.1 in order. The calculated features are stored in memory in an easy-to-control object, which is used for inferring with the model, or optionally they can be saved to a file on the hard disk for external use.

After the feature vectors have been calculated, the user can use the model to infer the importance scores of each point placed beforehand, using the *Evaluate with Model* button. This executes the steps mentioned in 3.4 using Sentis. Assuming execution finished with no errors, the *Place Predicted LP* can be used together with the threshold field to place the requested light probes in the scene, depending on their significance as calculated by the model. The algorithms 9 and 8 are used to complete this step of the process. the threshold field can be updated at any time to increase or decrease the amount of light probes placed.

At this point, the normal execution of the tool is complete, with the desired light probe

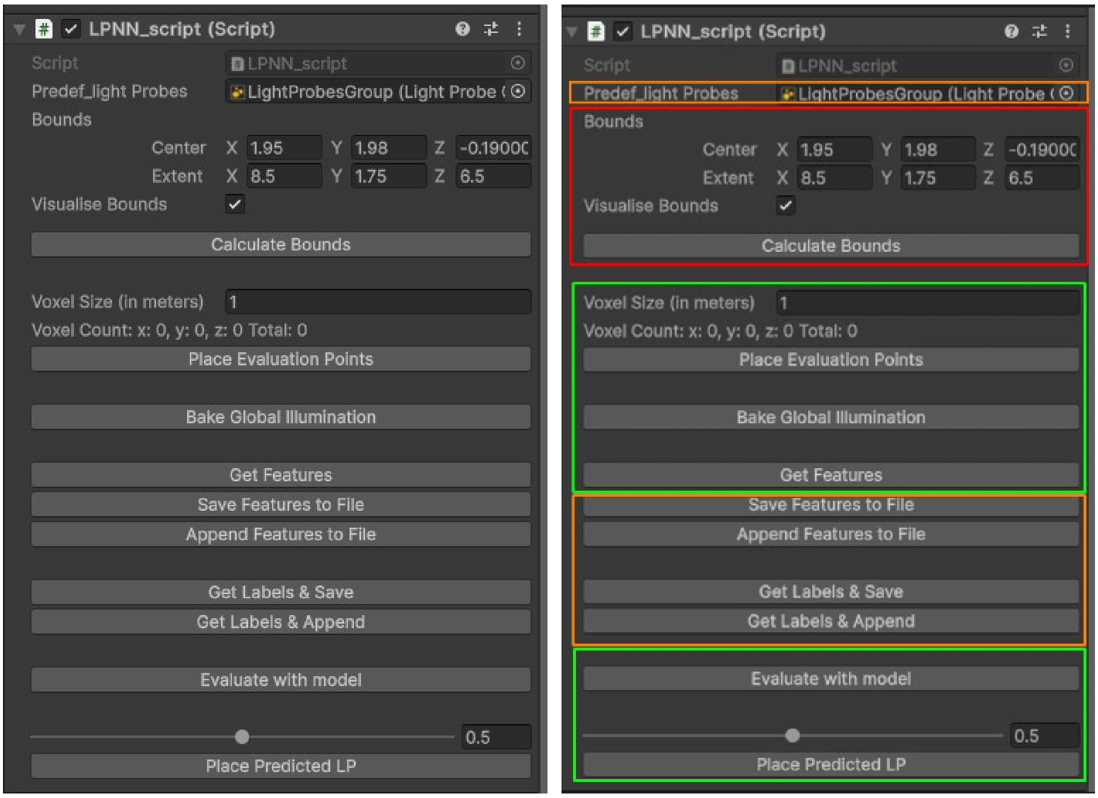

Figure 3.4: Overview of the UI of the tool, as seen inside the Unity Editor window. On the right, each section of the UI has been color-coded depending on the usage. *Red* are necessary pre-process steps. *Green* are steps required for normal usage, when a model has been trained. For manual retraining of the model, the *Orange* section houses the necessary fields.

layout placed. The positions are stored inside the *LightProbeGroup* component, which is located inside the *LPNN Parent* object.

3.5.2 Extraction of data for Retraining

However, the user is able to extract the features that the LPNN tool calculated, along with the label list, in order to be used externally for retraining of the model. This can be done using the fields shown in orange in figure 3.4. The bounds need to be calculated and the features vectors created for this process to be completed.

The user can click the *Save Features to File* button to extract the feature vectors into a .txt file on the hard disk. Multiple clicks of the button will replace all data already stored inside the file, replacing them with the current data. However, the *Append Features to File* button can be used to circumvent this, by appending the current features to the same file, keeping the old data intact, effectively generating a larger data-set. This allows for feature-collection from multiple Unity scenes and ground-truth light probe layouts, improving the model further.

Similarly, the *Get Labels & Save* and the *Get Labels & Append* buttons are responsible for calculating the labels for the current scene and saving or appending the data to a file on the hard disk. The process has been described in section 3.2. It is vital for the user to have a ground-truth light probe layout for the labels to be collected. The LightProbeGroup object can be exposed to the tool via the field shown in red in figure 3.4, called *Predef_light Probes*.

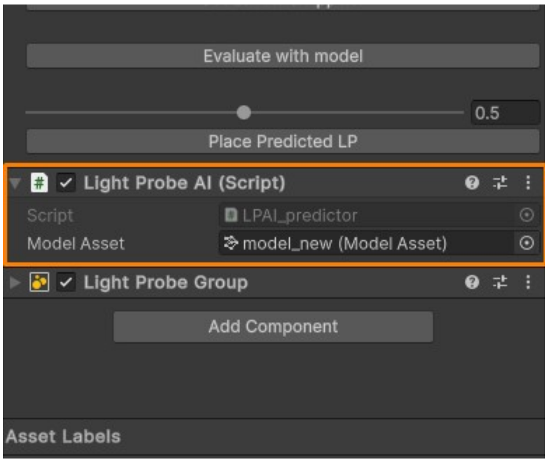

Figure 3.5: Figure showing the script that controls the AI model for inferring light probe importance in the scene, shown in *Orange*.

After extracting the necessary data, the user can now train an updated model externally using the process mentioned in section 3.3. For import back inside Unity for the tool to use the updated model, the user needs to pass in the new imported model to the field *Model Asset* of the *Light Probe AI* script, which is added automatically when creating the *LPNN Parent* object. The field can be seen in figure 3.5 in orange. After providing the model asset to the tool, standard workflow can be resumed, following the steps described in sub-section 3.5.1.

Chapter 4

Experiments

In this chapter, we present experimental results of the LPNN approach. The evaluation is qualitative, since the nature of light probes and their optimal placement is subjective to the user and the needs of the application. We focus mainly on speed of execution in relation to light probe layout given by the tool. We compare performance results to LumiProbes (Vardis, Vasilakis, and Papaioannou 2021b). All experiments were conducted in Unity 6 version 6000.0.38f1, on a system comprising of an NVIDIA RTX2060M GPU, 16GB DDR4 RAM and an Intel i7-9750H CPU, on a Windows 10 Operating System.

4.1 Performance

Since the focus of the thesis was to speed up the placement of light probes in this stage of any 3D application development iteration process, we will focus on the execution time of the tool and also critique the results and their qualitative properties shortly, to make sure the placements are correct. Indeed, as seen in table 4.1, the LPNN approach speeds up light probe placement by orders of magnitude faster than the LumiProbes, but it may suffer from occasional misplacement; probes that were placed in positions that are not vital, leading to oversampling, which we will show shortly.

In experiments where LumiProbes and LPNN were requested to place close to the same amount of light probes in the same scene, LPNN time stayed close to constant, typically up to a few seconds, regardless of the scenario or the amount requested. Results for various amounts of light probes and scenes are presented in table 4.1. Times are averaged over multiple runs. Units are represented in minutes (m), seconds (s), or milliseconds (ms). Where applicable, we also append the settings used for each tool. For LumiProbes, settings include the grid parameters and the evaluation-point count. All other settings are as follows: Evaluation point placement type is set to Poisson, Decimation type is set to Medium, Decimation directions are averaged, Decimation metric is set to Chrominance, Minimum LP set is disabled, and Maximum Error is set to 3. For LPNN, settings include the threshold value used for the specific result and the cell size of the 3D grid, in order.

Figures for the results are shown in section 4.2. Memory requirements for this tool are strictly dependent on the amount of light probes present in the scene, since all information needed by the tool is either already present in Unity and are needed regardless of the presence of the tool, e.g. Global Illumination data, or are stored on the Hard Disk of the system as files with storage sizes dependent on the amount of probes placed before the execution of the tool. File sizes for 150 probes were less than 2KB. Typically, the tool creates files needed for placing the light probes with size-scaling 1KB per approximately 100 probes.

For execution times greater than 20000 seconds (approximately 5 hours and 30 min-

Execution Results					
Method	Scene	Time	P. Count	P. Present (&Removed)	Settings
LumiProbes	Sponza	22.443s	105	34 (75)	(7,3,5), 128
		600.285s	240	54 (186)	(12,4,5), 256
		>20000s	420	——	(14,5,6), 256
	Office	51.059s	144	84 (60)	(12,3,4), 128
		919.134s	288	182 (106)	(12,3,8), 256
		>20000s	630	——	(14,5,9), 256
	Corridor	161.151s	180	120 (60)	(20,3,3), 256
		477.648s	243	147 (96)	(27,3,3), 256
Ours	Sponza	**5.3ms**	90	54 (36)	0.4, 2.0
		17.8ms	400	40 (360)	0.9, 1.3
	Office	**7.8ms**	140	34 (106)	0.758, 1.87
		25.2ms	832	117 (715)	0.859, 1.10
	Corridor	**10.9ms**	186	84 (102)	0.549, 1.94
		15.7ms	246	95 (151)	0.615, 1.50

Table 4.1: Execution time for LPNN and LumiProbes on a select number of scenes and probe counts. *P. count* represents the total amount of probes in the scene, before simplification. *P. Present* depict the final amount of probes after running the tools. The number in parenthesis is the amount of probes removed by the tool, in respect to the settings. Fastest times are shown in **bold**.

utes) the execution was forcibly stopped. Therefore, results for those occurrences are not collected, since for most scenarios and 3D scenes, the time needed for manual placement of light probes is shorter than 5 hours.

4.2 Quality

The tools were tested on the aforementioned edited Sponza scene (McGuire 2017), Corridor scene (Vardis, Vasilakis, and Papaioannou 2021a) and Office scene (CG AUEB 2021). We will present the qualitative results. As we will see shortly, LPNN correctly places light probes in areas of high variance.

4.2.1 Light Probe placement Quality

Before we can describe and analyze the layout provided by the tool, it is vital to describe what defines a high-quality light probe layout.

High-quality light probe placement optimally balances capturing lighting fidelity against the desired resource constraints. Light probes should be concentrated in regions where direct or indirect illumination exhibits rapid variation in space, such as near shadow edges, light occluders or areas with strong color bleeding. Placement of light probes in such locations is vital in order to minimize interpolation error of the irradiance information. Conversely, uniformly lit areas, such us open environments, probe density can be reduced to a minimum without perceptible loss of quality.

As seen in figure 4.1 and figure 4.2, the light probe layout depends highly on the surrounding context of the scene. By adapting probe density to lighting variance and geometric complexity, we ensure that the global illumination information received by dynamic objects in the scene is sufficient to reconstruct a close approximation of the true ground-truth solution, being path-tracing as mentioned in section 1.1.1, while placing the fewest probes possible depending on the resource limits.

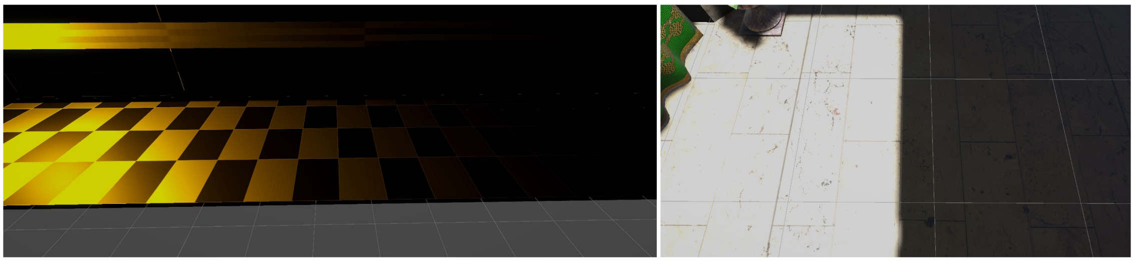

Figure 4.1: Comparison between a location of gradual transition of the illumination, compared to a more instantaneous change. Examples taken from Corridor scene (left) and Sponza scene (right).

A lower amount of light probes avoids long light baking times as well as reducing memory requirements in runtime. Additionally, if the probes are placed manually, a smaller amount of light probes makes the manual layout a process taking a fraction of the time required by a larger count. This enables artists to iterate rapidly, both in cases of manual placement and automatic placement tools, like the one proposed in this thesis.

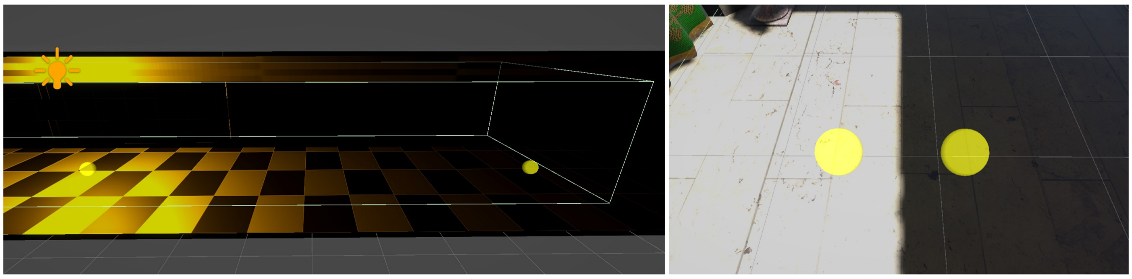

Figure 4.2: Scenes depicting locations of close-to-optimal light probe placement, depending on the context of the scene and the location currently shown. Worth noting is the fact that the amount of light probes depicted is not enough to created a working light probe group inside Unity, which requires four or more light probes to create a tetrahedron. The layout depicted is only for clarity reasons, and not application-ready. Examples taken from Corridor scene (left) and Sponza scene (right).

Equally important to the density of the probes is maintaining spatial uniformity and coverage of the scene. Light probes must be far apart enough that the developer avoids wasting memory and reduces bake time, but clustered enough that no interpolation errors are present; errors that produce visible seams in the lighting as objects move between the tetrahedral cells of the probes. A high-quality placement strategy enforces a range of ideal distance between light probes to avoid gaps or clusters.

4.2.2 Indoors Example

As shown in figure 4.4, we can clearly see that the LPNN tool has correctly placed light probes between the blue and green light sources, as well as between the green and red light sources, locations with great variance in Chrominance, It has additionally kept the amount of probes at a minimum, only placing one probe at each location mentioned. This result is close to optimal placement, since the distance between the light sources is only 4 units, making additional probes unnecessary in most scenarios. It should be noted that the tool did correctly place probes on the edges of the bounds, seen as light-green colored lines. This ensures that any dynamic object that moves outside the bounds set by the user continues to have some light-probes information for its illumination.

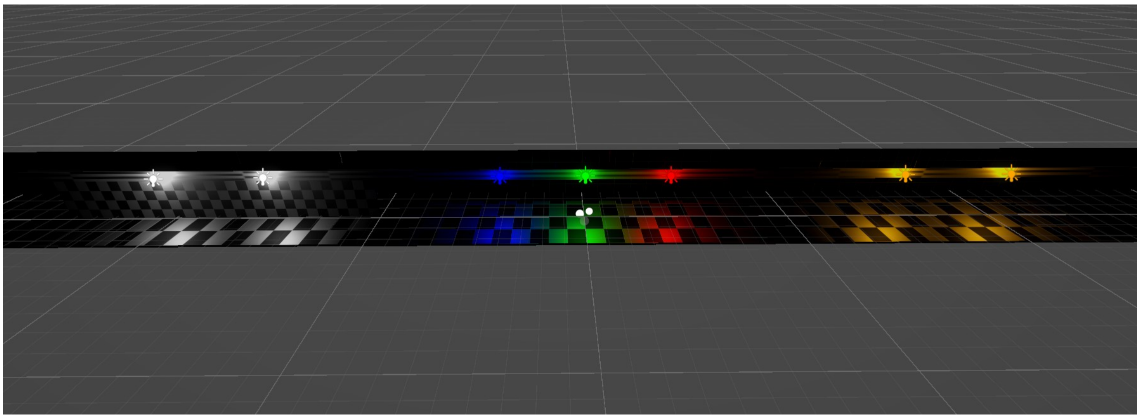

Figure 4.3: The Corridor scene (Vardis, Vasilakis, and Papaioannou 2021a) without any light probes placed. We show only the area of the scene that is inside the bounds of both LPNN and LumiProbes tools.

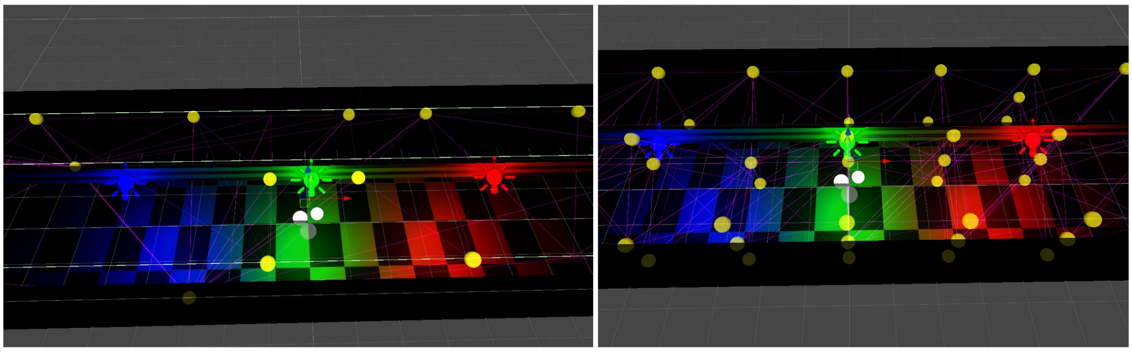

Figure 4.4: An indoor 3D Scene showing a comparison of light probe placement in a **high** color-variance scenario, between LPNN (left) and LumiProbes (right) with settings 0.549, 1.94 and (27,3,3), 256 respectively, in the Corridor scene.

Furthermore, in figure 4.5 we can see a similar result. The light probe between the two white light sources is vital. The areas left and right from the two light sources have light probes only in the dark areas, making the light transition when a dynamic object moves within this section of the scene smooth. Additionally, the light sources are next to an opaque wall, meaning no light present on the other side of the wall. Therefore, the model decided that placing light probes is only important on the edges of the wall, which is what we see in the example. It is important to note that on the left side of the corridor, as seen in the left image of figure 4.5, the model has placed a great amount of light probes, even though there is low light variance since that section of the scene is not illuminated.

Additionally, as seen in figure 4.7, the model suffers from under-sampling in certain scenarios; placing fewer light probes than necessary, resulting in partially incorrect lighting. However, this can be resolved in two immediate ways; the user can execute the tool again with different parameters, as seen in figure 4.8, or the user can manually place a small number of light probes in locations they deem vital. The tool has placed only 2 light probes on the lower section of the scene, resulting in elongated tetrahedrons and incorrect lighting for dynamic objects placed low. Additionally, the light probes placed on the dark side of the wall in figure 4.7 require one more row of light probes placed next to the wall, making any object inside that section of the scene completely dark due to the lack of light in that section. With the settings present in figures 4.4, 4.5, and 4.7, the tool doesn't have enough light probes in the desired locations to correct this issue. This can be fixed by

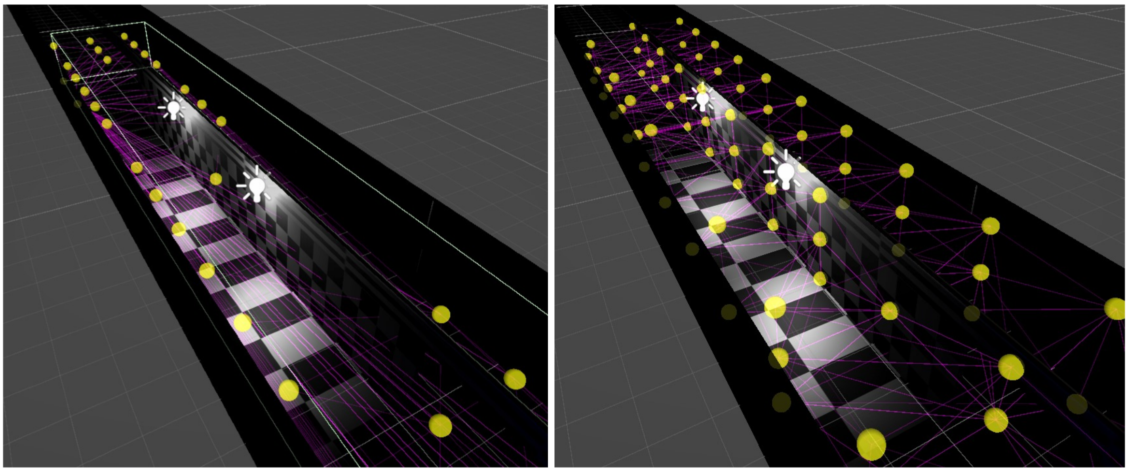

Figure 4.5: A 3D Scene showing a comparison of light probe placement in a **low** color-variance scenario, but **high** luminance variance, between LPNN (left) and LumiProbes (right) with settings 0.615, 1.5 and (27,3,3), 256 respectively, in the Corridor scene.

decreasing the cell size parameter, as seen in figure 4.6.

With better grid layout, the model correctly placed probes on both sides of the wall, correctly capturing the lighting information of the area. This can be seen in figure 4.6. On the left side of the wall, a number of probes were placed around the light sources, making any dynamic object that traverses the location have accurate illumination data. Similarly, on the right side of the wall, we can see probes placed flush with the wall, capturing the lack of light of the area, regardless of the proximity to light sources. This ensures that a dynamic object will remain unlit from the two light sources, if it is placed on the right side of the wall.

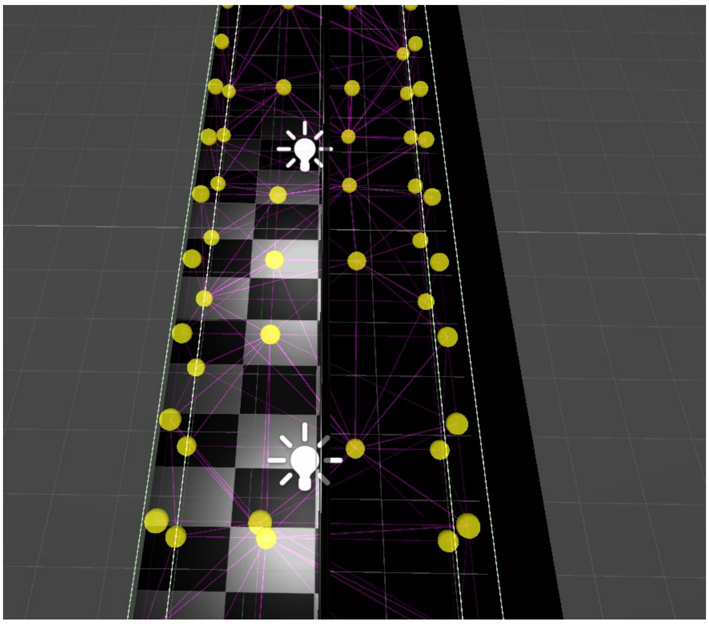

Figure 4.6: A 3D Scene showing a better placement of light probes in the Corridor scene, fixing the under-sampling issue of figure 4.5. The example was captured with LPNN with settings 0.244, 1.3.

It is worth noting that the model inferred that more points between the two light sources are important, placing three instead of one in figure 4.5. Even though this can be considered as slight oversampling, the amount of additional light probes is minimal.

This result can be recreated for figures 4.4 and 4.7 with the same steps as before; a decrease in the cell size for better sampling, with a finer-tuned threshold value. This will result in light probes being placed flush with the beam present on the top of the scene, or any other under-sampling scenario.

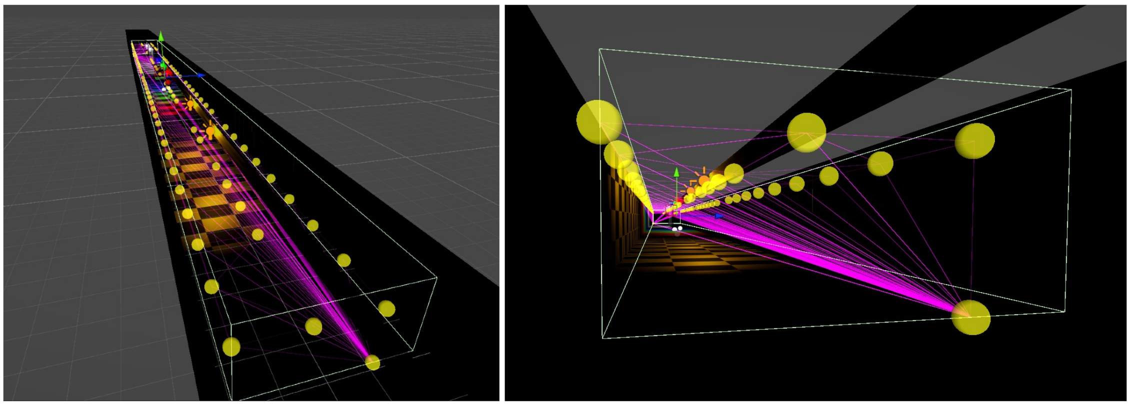

Figure 4.7: A 3D Scene showing under-sampling of light probe placement in the Corridor scene. The example was captured with LPNN with settings 0.615, 1.5.

As mentioned, with different settings, we can resolve the under-sampling issue from figure 4.7, as seen in figure 4.8. By increasing the cell size but reducing the threshold, the tool has placed more probes on the lower section of the scene, from just 2 to 8, leading to better results overall. This can be further increased with additional tuning of the settings, but it may lead to oversampling in certain high-variance locations elsewhere in the scene.

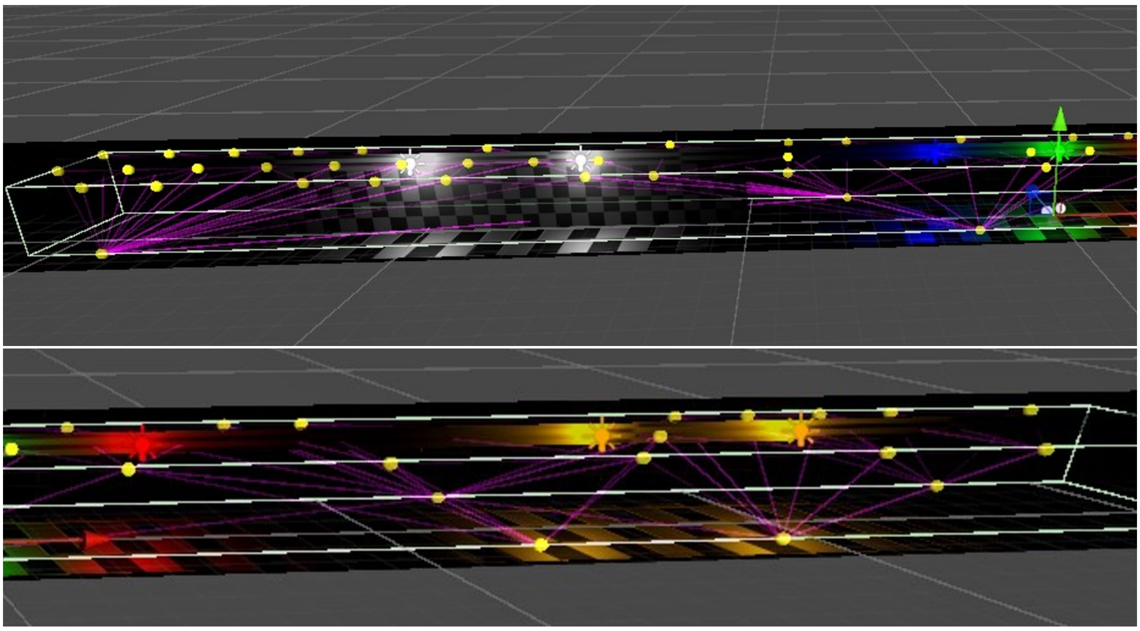

Figure 4.8: A 3D Scene showing improved light probe placement in the Corridor scene. The example was captured with LPNN with settings 0.549, 1.94.

4.2.3 Outdoors Example

So far, only results of indoors scenes have been shown and described. Indoor scenes are ones where all the illumination provided by the light sources, (lamps or the outdoor light leaking through the windows of a room) is contained within an area bounded by opaque surfaces like walls. However, in addition to indoor scenes, outdoor scenes are equally as important in a lot of 3D applications. The Sponza scene (McGuire 2017) is an outdoors environment, where illumination is provided by a light source closely imitating a sun. The presence of walls without a ceiling, together with the sun angle being not-vertical to the floor, leads to interesting shadows and light interactions within the objects. As we will see shortly, LPNN speeds up the process of placing light probes in the scene significantly, requiring only minimal manual tweaking.

Figure 4.9: The Sponza scene (McGuire 2017) without any light probes placed. We show only the area of high importance, where the light probes should be placed.

Seen in figure 4.10, the LumiProbes tool has placed several light probes on the two sides of the shadow created by the angled sun direction. This ensures that a dynamic object traversing the floor of the scene will receive GI information from the light probes, making the transition between the illuminated side on the right and the shadowed side on the left smooth.

At first glance, the layout proposed by the LPNN tool for the settings mentioned in the caption of the figure, seems incorrect. The edge between the illuminated and the shadowed sides lacks light probes, resulting in incorrect illumination for any dynamic object traversing the scene. However, even though the initial placement is sub-optimal, with minimal manual tweaking this can be improved significantly.

In figure 4.11 we can see the resulting layout, after manually translating the light probes a few units to the right. This process comprises of a small number of inputs required by the user, taking a few seconds to complete. The speedup of the tool is still present, even if it requires a few manual tweaks, saving time for the developer during this part of the process when developing 3D applications. In the improved layout seen in figure 4.11 on the right, the light probes are placed on both sides of the edge created by the angled sun illumination being occluded by the walls of the building.

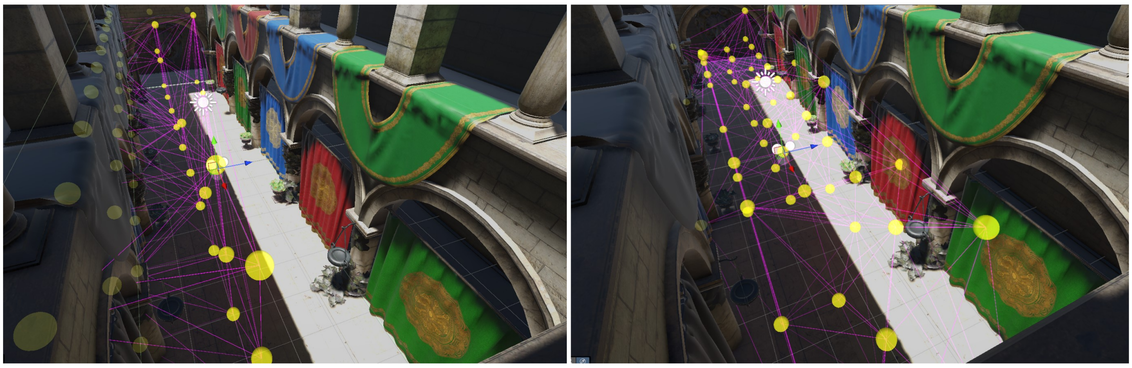

Figure 4.10: A 3D Scene showing a comparison of light probe placement between LPNN (left) and LumiProbes (right) with settings 0.4, 2.0 and (12,4,5), 256 respectively, in the outdoor Sponza scene.

Figure 4.11: A 3D Scene showing a comparison of the default light probe placement of the LPNN tool (left) and after translating the LightProbeGroup object a small number of units to the right (right).

Additionally, the tweaked layout places light probes in front of the colored surfaces on the right wall of the scene. A dynamic object that requires indirect lighting will correctly be colored lightly by the colored materials of the surfaces, and the color will also be interpolated when traversing between them. Shown closer in figure 4.12, we can clearly see that light probes are placed only in front of the colored materials. Taking into consideration the context of the scene, placing light probes at every grid location on that close-up area results to oversampling, placing more probes than needed, increasing the memory usage of the scene. Therefore, placing a probe between the blue and red surfaces is unnecessary, something that the LPNN model has correctly avoided.

However, it has still dictated that a light probe between the nearby red and greed surfaces is required. With the same reasoning, we can determine that it is not vital for a light probe to be present in that location. The indirect illumination from the surfaces will be soft enough to not warrant a dense light probe layout at each location. Additionally, the edge of the shadowed area is still covered by the light probes around the mentioned position of the unnecessary light probe, since the transition between the illuminated and shadowed areas stays consistent through the scene. We decided to not manually remove the mentioned light probe for the sake of completeness, however doing so is a trivial process, comprising of only a couple of user inputs, adding only a few seconds to the total time needed for this step of the process of development.

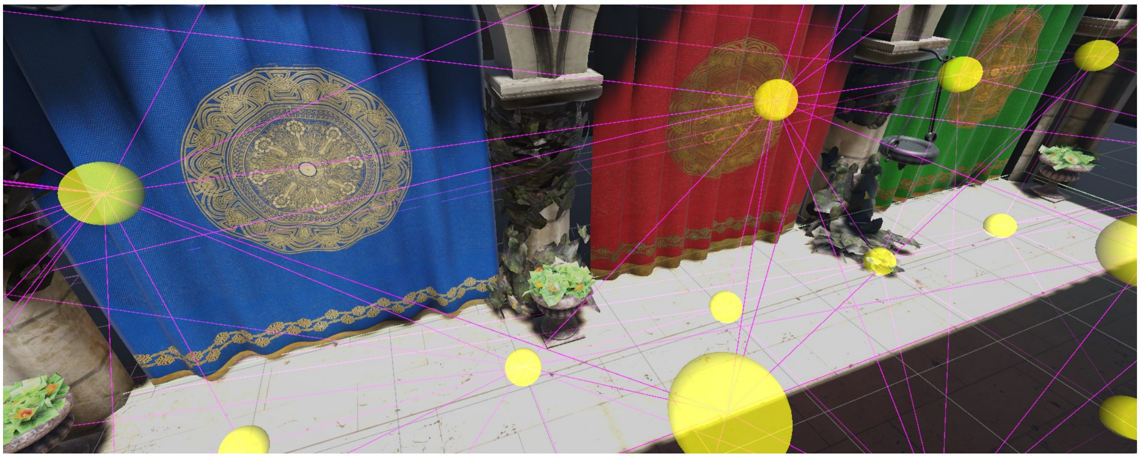

Figure 4.12: Detailed view of the tweaked layout shown in figure 4.11, showing light probes being placed in front of the colored materials, able to capture indirect illumination data for dynamic objects moving between them.

Similar results were present regardless of the settings used for LPNN. The translation tool was required often, but not always. We chose to show an example using settings that best displayed our desired layout, ensuring the same number of light probes were placed with LumiProbes. The settings for both tools can be seen in the caption of figure 4.10, or in table 4.1.

4.2.4 Shortcomings of LPNN

It is important to note that the light probe layout described in sub-section 4.2.3 is inherently sub-optimal, which impacts the accuracy of dynamic object illumination. Any harsh edge in illumination, an example of which is present in the Sponza seen shown in the same sub-section, requires a tightly-packed light probe layout surrounding it, in order to ensure that a dynamic object has an instant transition between the different illuminations.

A sparser grid like the one used in the same sub-section will correctly place light probes on both sides of the light-edge, however the distance between them will make the transition smooth, resulting in inaccurate GI on the surface of a dynamic object. This issue cannot be resolved by the model, as it originates from the cell size chosen by the user, highlighting the importance of user input in achieving optimal results. Even with a theoretical perfect model, faulty user usage of the tool places the model in a state that it can only minimize the user error as best as it can. It can not resolve it, making user error one of the largest factors of sub-optimal light probe placement using the LPNN tool.

The LPNN model used during the creation of this thesis can be improved. In rare occasions, the inferred importance scores fail to cover areas of high GI variance. As seen in figure 4.14, the light probes placed by LPNN (left) fail to completely capture the lighting information of the three colored light sources in one room. Additionally, no light probes were placed in the left of the scene, near the white light source. Any dynamic object in this scene will receive insufficient illumination data, resulting in incorrect surface color.

Additionally, in the same scene, we can see that the right half of the building contains multiple light probes, even in areas where the change in illumination is minimal or non-existent. Not all light probes are unnecessary. A dynamic object in this part of the scene will have sufficient data from the light probes surrounding it, therefore the surface of the object will be shaded correctly. However, due to the lack of significant illumination

Figure 4.13: The Office scene (CG AUEB 2021) without any light probes placed.

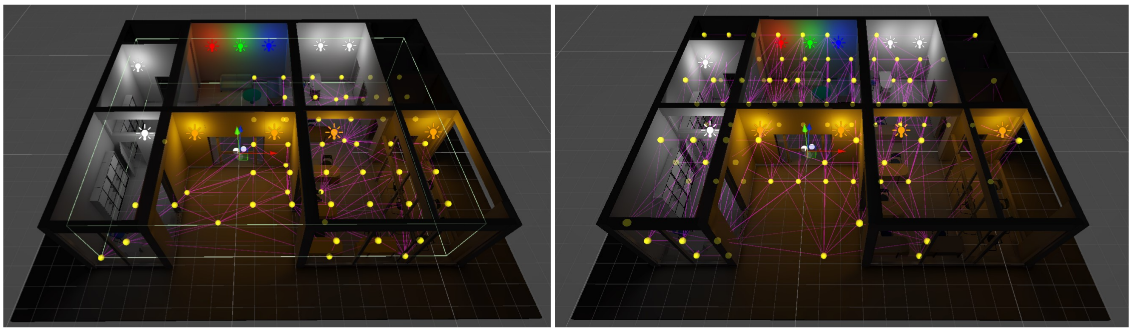

Figure 4.14: Overview of faulty light probe placement inferred by the model. Settings used are 0.758, 1.87 for LPNN (left) and (12,3,4), 128 for LumiProbes (right).

changes, the amount of placed light probes can be reduced significantly, while keeping the same visual quality. The currently oversampled area results in increased memory usage with no improvements in the quality of GI on dynamic objects.

This problem results from the model itself. It is important to have sufficient amount of high quality data during training; data from a variety of scenes and examples, including many edge-cases and variations. We trained the model using 3485 total feature vectors from different scenes, as detailed in section 3.3. However, a bigger dataset can improve the accuracy of the model, potentially avoiding the issue mentioned. In cases like the one described, it is necessary that the user attempts using different settings for the tool and potentially requiring an increased manual tweaking process to provide the desired results.

With different settings, mainly a much denser EP grid, the importance scores inferred by the model improve slightly. Even though they are still suboptimal, the light probes are located in areas of variance more often, but still failing to appear in all and with proper layout. As seen in figure 4.15, the model successfully placed light probes around the three differently-colored light sources in the central room, and additionally placed probes on the white light source to the left of the room. Furthermore, a few light probes were placed on the dark rooms on the two corners of the building, ensuring a dynamic object will be properly shaded when entering those areas.

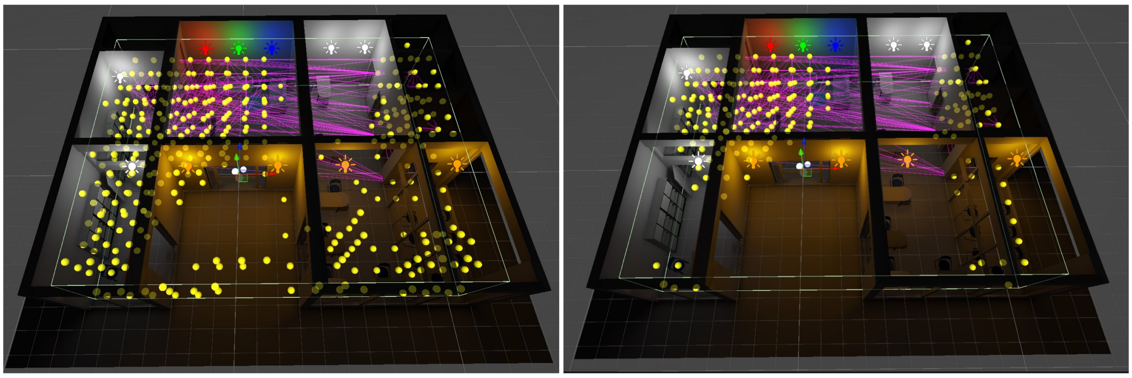

Figure 4.15: Overview of faulty light probe placement inferred by the model. Settings used are 0.611, 1 (left) and 0.785, 1 (right).

However, it is apparent that this layout is still not sufficient. The large amount of light probes placed, even with an increased threshold, as seen in the same figure 4.15, results in much higher memory usage during runtime, greatly impacting the performance of the application. Furthermore, in an attempt to reduce the number of light probes by increasing the threshold, keeping only the most important light probes, we can see a significant gap in light probe placement in areas of high variance. Namely, on the image on the right of figure 4.15, the rooms on the bottom of the figure, as well as the room with the two white light sources lack significantly in light probe coverage. Any dynamic object traversing those rooms will receive incorrect lighting, immediately visible to the user.

Furthermore, the dense grid and consequently the large amount of light probes placed makes manual tweaking by the developer a very time-consuming task. It is natural for the developer to require the most optimal layout for the present scene, but relocating each light probe in this current scenario will take approximately the same amount of time than placing the light probes manually.

Even though the two examples shown in figures 4.14 and 4.15 are from the same scene, with the only difference being the size of the cell which results in a denser grid, the innate uncertainty of a Neural Network model results in an inferred output that can on occasion be faulty or incorrect, as presented. Additional tweaking of the cell size and the threshold value can create an improved layout for the light probes, as seen in figure 4.16.

The new layout has placed more light probes on the areas mentioned before, specifically those that contained minimal or none at all. The rooms on the right of the building have enough light probes to capture the transition between the colored lights and the white lights, the orange lights and the white lights as well as the white light sources and the dark rooms on the right corner of the building. This resolves most issues with the previous layout, as well as reducing the density of the grid and therefore the amount of light probes present in the scene.

However, it is important to mention that the layout present in figure 4.16 has been tweaked manually, and is not the default result inferred by the LPNN model with the setting specified in the caption of the figure. We manually tweaked the layout by removing a small number of light probes from densely populated areas, since their impact to the accuracy of lighting information on a dynamic object is not vital. No other manual tweaking was done; the positions of light probes that were not removed is the same, and no light probes were added to the grid and placed manually. The manual process was completed within 5 minutes from the moment the model placed the initial layout of light

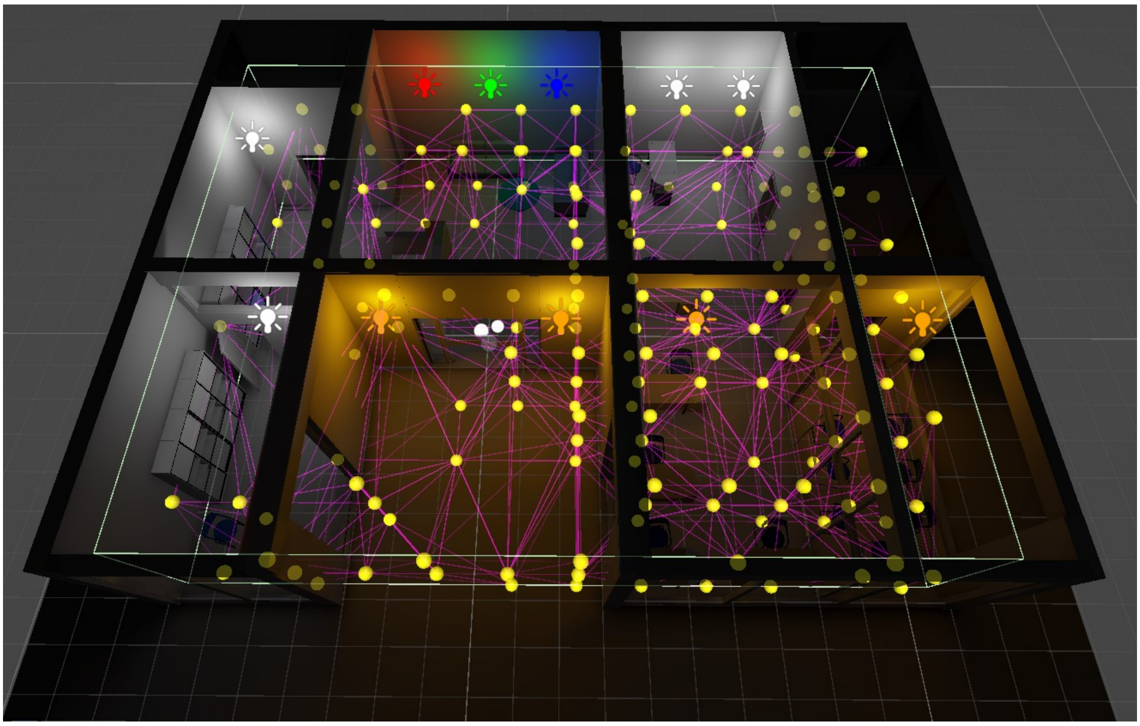

Figure 4.16: Overview of an improved light probe placement inferred by the model. Small manual tweaking was done to remove a small number of light probes. Settings used are 0.35, 1.49.

probes with the provided settings. In total, from the moment we first opened the scene in Unity to the moment of capturing the result, the time spent was close to 10 minutes of real time. Even though manually placing the light probes is a process that requires much more time, especially on larger scenes, the results present can still be improved upon to approach the theoretical optimal even more.

We can clearly see that the process of finding the best layout by experimenting with the settings is a process that can on occasion result to sub-optimal light probe layouts, requiring manual tweaking or ,on rare occasions, even restart of the process. Work can be done to improve the model and the tool, as described in section 5.

Chapter 5

Conclusions and Future Work

In this thesis, a method is implemented to accelerate the placement of light probes in a 3D scene during the development process. The method is experimented on, and the results are presented. The LPNN method and its normal usage are described. Additionally, the process of collecting the required features and labels needed for retraining the Neural Network is also presented. This approach assigns an importance score to a grid-like layout of Evaluation Points, which are then used to place light probes in the scene, depending on a variable threshold value controlled by the developer. Finally, quantitative and qualitative results are presented.

In the experiments conducted, we concluded that the Neural Network approach is capable of reducing the time needed by the developer for a sufficient light probe layout, as well as minimizing the amount of manual tweaking required for optimal results. The results present a solution that approaches a theoretical optimal light probe layout. While not free of oversampling or undersampling, the LPNN tool provides a consistently suitable layout within a fraction of the time needed by other methods, including the manual approach.

Moving forward from this thesis, an improved Neural Network architecture can be explored, implemented and experimented on. An approach that takes into consideration the edges of each light probe can potentially improve the accuracy of the model presented. While a Deep Learning approach requires a large dataset to be able to generalize sufficiently, an improved architecture and implementation can reduce the cost and improve accuracy by providing more variety in the training data. Additionally, a heuristic approach can be combined, removing or adding light probes in areas that the model inferred to be of low-significance, allowing for results closer to the theoretical optimal layout. Lastly, a new approach can be explored that combines the LPNN approach with the Neural Light Field Probes method (You, Geiger, and Chen 2024), ensuring that the contributions of both methods are preserved.

Bibliography

CG AUEB, Group (2021). *Office Unity Scene*. URL: http://graphics.cs.aueb.gr/graphics/index.html.

Crassin, Cyril et al. (2011). "Interactive indirect illumination using voxel-based cone tracing: an insight". In: *ACM SIGGRAPH 2011 Talks*. SIGGRAPH '11. Vancouver, British Columbia, Canada: Association for Computing Machinery. ISBN: 9781450309745. DOI: 10.1145/2037826.2037853. URL: https://doi.org/10.1145/2037826.2037853.

EpicGames (2025). *Lumen GI*. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/lumen-global-illumination-and-reflections-in-unreal-engine.

Greger, G. et al. (1998). "The irradiance volume". In: *IEEE Computer Graphics and Applications* 18.2, pp. 32–43. DOI: 10.1109/38.656788.

Guo, Jie et al. (Nov. 2022). "Efficient Light Probes for Real-Time Global Illumination". In: *ACM Transactions on Graphics* 41, pp. 1–14. DOI: 10.1145/3550454.3555452.

ITU-R (June 2015). *Parameter values for the HDTV standards for production and international programme exchange*. URL: https://www.itu.int/dms_pubrec/itu-r/rec/bt/R-REC-BT.709-6-201506-I!!PDF-E.pdf.

Kajiya, James T. (Aug. 1986). "The rendering equation". In: *SIGGRAPH Comput. Graph.* 20.4, 143–150. ISSN: 0097-8930. DOI: 10.1145/15886.15902. URL: https://doi.org/10.1145/15886.15902.

Lafortune, E (1996). "Mathematical Models and Monte Carlo Algorithms for Physically Based Rendering". URL: https://web.archive.org/web/20160617231944/http://www.graphics.cornell.edu/~eric/thesis/index.html.

Lafortune, Eric P. and Yves D. Willems (1993). "Bi-directional path tracing". In: *Proceedings of Third International Conference on Computational Graphics and Visualization Techniques (Compugraphics '93)*. Alvor, Portugal, pp. 145–153. URL: https://graphics.cs.kuleuven.be/publications/BDPT/.

McGuire, Morgan (2017). *Computer Graphics Archive*. https://casual-effects.com/data. URL: https://casual-effects.com/data.

McGuire, Morgan et al. (2017). "Real-time global illumination using precomputed light field probes". In: *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D '17. San Francisco, California: Association for Computing Machinery. ISBN: 9781450348867. DOI: 10.1145/3023368.3023378. URL: https://doi.org/10.1145/3023368.3023378.

Nvidia (2024). *RTXGI Dynamic Diffuse GI.* URL: https://github.com/NVIDIAGameWorks/RTXGI-DDGI.

Qi, Charles R. et al. (2017). *PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation.* arXiv: 1612.00593 [cs.CV]. URL: https://arxiv.org/abs/1612.00593.

Teuber, Maurice et al. (2024). "Geometry-Based Optimization of Light Probe Placement in Virtual Environments". In: *2024 International Conference on Cyberworlds (CW)*, pp. 9–16. DOI: 10.1109/CW64301.2024.00011.

Unity (2016). *Light Probes Example.* URL: https://docs.unity3d.com/530/Documentation/Manual/class-LightProbeGroup.html.

— (2025). *Adaptive Probe Volumes.* URL: https://docs.unity3d.com/Packages/com.unity.render-pipelines.high-definition@17.0/manual/probevolumes-concept.html.

Vardis, Konstantinos, Andreas Vasilakis, and Georgios Papaioannou (2021a). *Corridor Unity Scene.* URL: https://github.com/cgaueb/light_probe_placement/tree/main/Light%20Probes/Assets/Scenes/TestProbesScene.

— (May 2021b). "Illumination-driven Light Probe Placement". In: DOI: 10.2312/egp.20211026.

Veach, Eric and Leonidas J. Guibas (1997). "Metropolis light transport". In: *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques.* SIGGRAPH '97. USA: ACM Press/Addison-Wesley Publishing Co., 65–76. ISBN: 0897918967. DOI: 10.1145/258734.258775. URL: https://doi.org/10.1145/258734.258775.

Wang, Yue et al. (2019). "Fast non-uniform radiance probe placement and tracing". In: *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games.* ACM, p. 5.

Wikipedia contributors (2025). *Lightmap — Wikipedia, The Free Encyclopedia.* [Online; accessed 15-June-2025]. URL: https://en.wikipedia.org/w/index.php?title=Lightmap&oldid=1285782125.

Xu, Yang et al. (2022). "Precomputed Discrete Visibility Fields for Real-Time Ray-Traced Environment Lighting." In: *EGSR (ST)*, pp. 81–92. ISBN: 978-3-03868-187-8. DOI: 10.2312/sr.20221158.

You, Zinuo, Andreas Geiger, and Anpei Chen (2024). *NeLF-Pro: Neural Light Field Probes for Multi-Scale Novel View Synthesis.* arXiv: 2312.13328 [cs.CV]. URL: https://arxiv.org/abs/2312.13328.