Extending PRT Framework for Lowly-Tessellated and Continuous Surfaces

Thesis submitted in partial fulfillment
of the requirements for the degree of

Masters of Science
in
Computer Science and Engineering by Research

by

Dhawal Sirikonda
2019201089
dhawal.sirikonda@research.iiit.ac.in

International Institute of Information Technology
Hyderabad - 500 032, INDIA
June, 2023

International Institute of Information Technology
Hyderabad, India

CERTIFICATE

It is certified that the work contained in this thesis, titled "Extending PRT Framework for Lowly-Tessellated and Continuous Surfaces" by Dhawal Sirikonda, has been carried out under my supervision and is not submitted elsewhere for a degree.

_____ _____

Date Adviser: Prof. PJ Narayanan

To, the invested individual

Acknowledgments

This master's thesis and my journey through it resulted from the efforts of many people who constantly supported me at different points in time.

Firstly, I want to thank my advisor PJ Narayanan, my brother Utkal Sirikonda, and my brother from another mother, Pratikkumar Bulani, for their undying support during some of the most enervating times at IIIT-H. The initial two years of my master's at IIIT were greatly disturbed and unproductive due to the pandemic.

I want to thank my advisor, Prof. PJ Narayanan, for believing in me and taking the risky decision of allowing me to switch tracks from M.Tech to MS by Research under his supervision. His attitude to research and his solid understanding of traditional graphics and vision pipelines have changed me in ways I could never have imagined. Someday I would like to establish an institutionalized lab as he did and nurture enthusiastic youngsters in their journey. I also want to thank Prof. Avinash Sharma, who has motivated me to explore CVIT since I started as an M.Tech student, laying the initial foundation for this thesis. I would also like to thank him for providing informative feedback on various aspects of life and being a research guide in the later stages of my master's.

I would like to thank Aakash KT for playing the role of an *"always present"* ever-helpful friend in introducing me to the field of rendering and helping me write my initial drafts for the conference. I would also like to thank PulkitRega for throwing so many papers (virtually) at me and teaching me how to get a quick overview of every work, as well as for his skill at doing research on Twitter. I would also like to thank Dr. Shah, Dr. Sakurikar, and Dr. Saini (soon to be) for providing wider perspectives about fields related to graphics, specifically computational photography, which will be my field of study for my doctoral journey at Dartmouth.

I can't stress enough how significant a role my friends played during this journey (chronologically): Raghu and Vineel (bachelors), Upinder, Pradeep, Manik, Utsav, Anupam, (the forever preachy) Sarkar, Astitva, Dr. Sagar, Sarath, Richa Misra, Sai, Ritam, Bala, SharanDeep, Shantika, Pramod and Rohan, for all the fun at CVIT and IIIT. I am also happy to have worked with new blood, Rahul, Ishaan, Kunwar, and Amogh, who reminds me of my naive, younger self but are much more talented.

Lastly, I would like to thank my parents, my extended family, Samuel and Praharsha, and my late maternal grandparents for believing in my decisions and supporting my journey.

Abstract

Precomputed Radiance Transfer (PRT) is widely used for real-time photorealistic effects. PRT disentangles the rendering equation into *transfer* and *lighting*, enabling their precomputation. Transfer accounts for the cosine-weighted visibility of points in the scene, while Lighting is usually a distant emitted lighting, e.g., environment. Transfer computation involves tracing several rays into the scene from every point on the surface. For every ray, the binary visibility is calculated, and a spherical function is obtained. The spherical function is projected into Spherical Harmonic(SH) domain. SH is a band-limited representation of spherical functions, and the order of SH decides the representation capacity of the SH (the higher the SH order better the approximation of a spherical function). The SH domain also facilitates fast and efficient integral computation by simplifying the integral into simple dot products and convolutions. The original formulation of PRT by Sloan et al. 2002 provides different storage requirements for the transfer—vectors in the case of diffuse materials and matrices in the case of glossy materials. Using matrices for Transfer representation makes it infeasible as the SH orders increase. The work of Triple Product Formulation by Ng et al. in 2004 extended the formulation to allow simple vector-based Transfer storage even for the case of glossy materials. Prior art stored precomputed transfer in a tabulated manner in vertex space. These values are fetched with interpolation at each point for shading. Since the barycentric interpolation is finally employed to calculate the final color across the geometry apart from the vertex locations, the vertex space methods require densely tessellated mesh vertices to obtain accurate radiance. Sometimes high-density(tessellated) meshes adversely affect runtimes and memory requirements. This is mainly observed in simple geometries with no additional detailing but still demanding higher triangle counts (e.g., planes, walls, etc.). The first work provides a solution by leveraging Texture space, which is more continuous than the Vertex space. We also added additional functionality to obtain inter-reflection effects in the texture space.

While Texture space methods provide faithful results in meshes, they require non-overlapping, area-preserving UV mapping, and a high-resolution texture to avoid artifacts. In the subsequent work, we propose a compact *transfer* representation that is learnt directly on scene geometry points. Specifically, we train a small multi-layer perceptron (MLP) to predict the transfer at sampled surface points. Our approach is most beneficial where *inherent mesh storage structure and natural UV mapping are unavailable*, such as Implicit Surfaces, as it learns the transfer values directly on the surface. Using our approach, we demonstrate real-time, photorealistic renderings of diffuse and glossy materials on SDF geometries with PRT.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Rendering has been of particular interest to production houses of movies and video games, both photorealistic and non-photorealistic renderings. While non-photorealistic rendering concentrates on a wide spectrum of artistic choices requiring manual intervention and validation, photorealism is guided by the physical modeling of light transport. In this work, we only concentrate on Photorealistic rendering. Rendering broadly deals with generating pixels stacked as 2D arrays forming an image. In the real world, the image captured by a camera collects the incoming light energies onto the sensor array. The sensor information is captured and processed to obtain an image.

In the case of computer-generated imagery(CGI), the task is usually the inverse of image capture using the camera. The scene has authored both geometry and lighting. A virtual camera is placed, and suitable material properties are assumed for the geometric content present in the scene. Panning the camera around creates the desired set of images. For the formation of this image, the methodologies used are categorized into two categories: Rasterization and Path Tracing.

1.1 Image Formation Methodologies

1.1.1 Rasterization Graphics Pipeline

Rasterization is the process of converting vector graphics or 3D models into a raster or bitmap image. In computer graphics, rasterization is an important step in the rendering pipeline. It involves converting a set of mathematical equations that describe a 2D or 3D object into pixels or dots that can be displayed on a screen.

The process of rasterization involves several steps. First, the object is broken down into individual polygons, which are then projected onto a 2D plane. Next, the pixels that are covered by each polygon are determined, and the color or texture of each pixel is calculated based on the properties of the polygon.

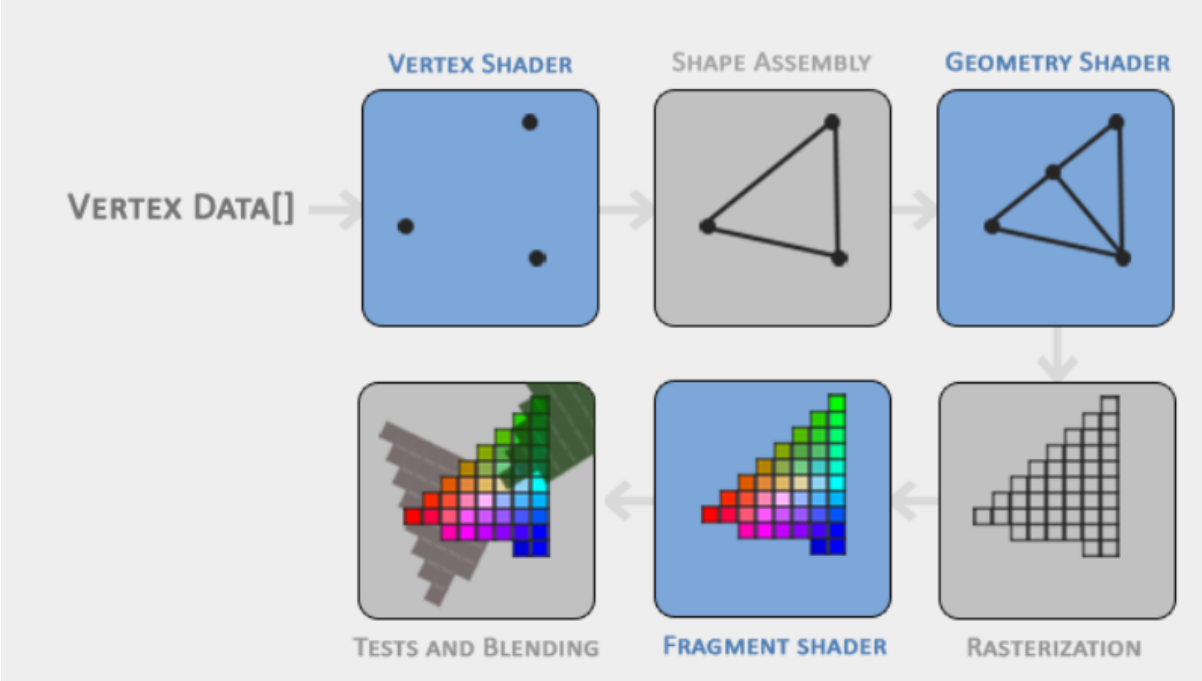


Figure 1.1: Rasterization Graphics Pipeline (image courtesy from learnopengl.com)

1.1.2 Ray Tracing Graphics Pipeline

Ray-tracing is a rendering technique used in computer graphics to create realistic images of 3D scenes. Unlike rasterization, which projects polygons onto a 2D plane, ray tracing simulates the behavior of light in a 3D environment, calculating the path of light rays as they interact with objects in the scene.

Ray tracing works by tracing the path of light rays from a virtual camera through each pixel in the image plane and calculating the color and intensity of each pixel based on the interactions of the light rays with the objects in the scene. This involves simulating the reflection, refraction, and absorption of light as it interacts with surfaces in the scene, as well as the effects of shadows and global illumination.

One of the advantages of ray tracing is that it can produce highly realistic images with accurate lighting and shadows. This makes it well-suited for creating images of complex scenes, such as those found in architectural visualization, product design, and film animation. Ray tracing can also produce images with high levels of detail and visual fidelity, making it a popular choice for creating photorealistic images.

However, ray tracing is computationally intensive and requires significant processing power and time to render images. To address this, various optimizations have been developed, such as using specialized hardware, parallel processing, and adaptive sampling techniques. The development of real-time ray tracing has also made it possible to use ray tracing in interactive applications, such as video games and virtual reality.

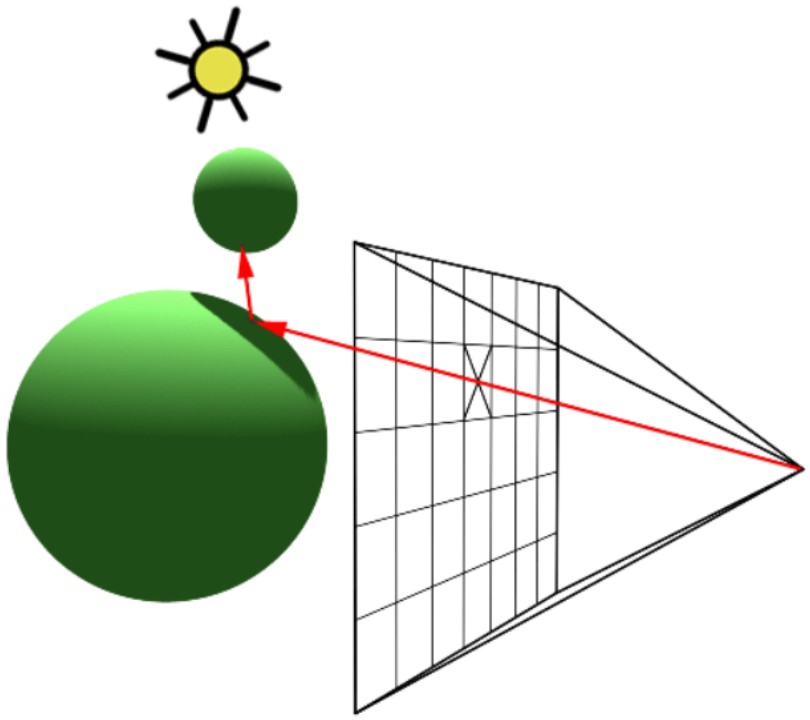


Figure 1.2: Raytracing Pipeline (image courtesy scratchapixel)

While these image formation methodologies exist, there are three fundamental aspects to consider for the formation of the image:

1. *How is the scene geometry represented?* (Sec. 1.2)
2. *How is the lighting of the scene defined?* (Sec. 1.3)
3. *What is the material property of the scene?* (Sec. 1.4)

1.2 Scene Representation

There are various ways of representing the scene geometry. Some of the few are discussed here. Fig. 1.3 shows a respective depiction of the geometries.

1.2.1 Meshes

A mesh is a 3D representation of an object or surface consisting of a set of vertices and faces. The vertices define the position of points in 3D space, while the faces connect the vertices to form triangles, quads, or other polygons. The topology of a mesh defines the structure of the connections

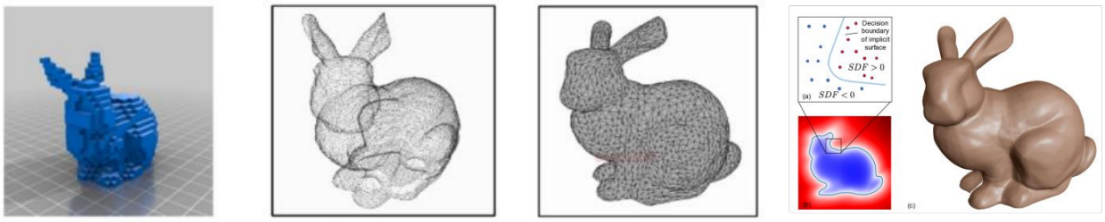


Figure 1.3: Geometric Representation (image courtesy Slides Stanford University)

between vertices and can affect the appearance and properties of the mesh. Meshes are commonly used in computer graphics, animation, and simulations and can be rendered using various techniques such as rasterization or path tracing.

Additionally, meshes also ship with hierarchical data arrangements, like vertices, faces, and UV parameterizations. These act as data storage techniques for rendering related tasks.

1.2.2 Point Cloud

A point cloud is a set of data points in a 3D space that represents the surface geometry of an object or scene. Each point in the point cloud corresponds to a specific location in 3D space and is typically associated with additional information such as color, intensity, or reflectivity. Point clouds can be generated from various sources, such as 3D scanners or LIDAR sensors, and can be used in a wide range of applications such as 3D modeling, virtual reality, robotics, and autonomous vehicles. However, point clouds can be computationally expensive to process and analyze due to their large size. Moreover, they are not commonly used for rendering purposes.

1.2.3 Voxel

Voxel-based representations are commonly used in medical imaging, where a three-dimensional image is generated by stacking a set of two-dimensional images taken at different depths. Voxel-based representations can also be used in computer graphics, where they can represent the shape of a 3D object or scene. Voxel data can be processed and manipulated in various ways, such as filtering or smoothing, and can be converted into other representations, such as point clouds or meshes.

1.2.4 Implicit Surfaces

Implicit surfaces are a mathematical representation of a 3D object or surface that defines the surface as the zero-level set of a scalar function. The scalar function takes a 3D point as input and outputs a value, which can be interpreted as the distance of the point from the surface of the object. The

implicit surface can be described by the equation $f(x, y, z) = 0$, where f is the scalar function. The sign distance function is a variation of the scalar function used to represent implicit surfaces. The sign distance function assigns a positive or negative sign to each point in space depending on whether it is inside or outside of the object represented by the implicit surface. The magnitude of the sign distance function is the distance of the point from the surface. The sign distance function can be used to create a signed distance field, which can be used in various applications such as collision detection, rendering, and shape manipulation. Implicit surfaces have some advantages over other representations, such as meshes or point clouds. They can represent complex and continuous shapes without explicitly defining the topology or structure of the object. Implicit surfaces can also be easily manipulated using mathematical operations such as blending or deformation. However, implicit surfaces can be tedious to the author as the complexity of geometry increases.
Of the aforementioned geometric representations, the point cloud and voxels are not commonly used in rendering tasks. This is due to the fact that rendering usually considers a surface and assigns a proper color to it at every point, while on the other hand, a point cloud consists of points that are infinitesimally small. This leaves out meshes and implicit surfaces. Both of these representations are widely used in the industry specifically in gaming and films. While meshes are commonly used to represent a vast majority of geometric content like the exteriors and interiors of buildings, human body models, etc, implicit surfaces are commonly used for modeling geometries with high detail and easy mathematical representation: terrains, tweakable geometries supporting the union, and intersection operations.

In this thesis, we concentrate on the *Meshes* and *Implicit Surfaces, specifically SDF* based geometries for the task of rendering.

1.3 Lighting

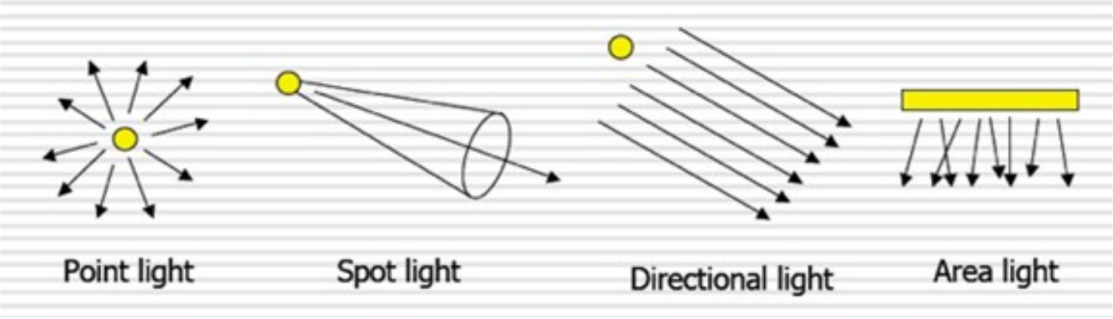


Figure 1.4: Light Sources (image courtesy Lecture Slides of University of Sulaimani)

There are several types of lighting commonly used in computer graphics, including:

- Ambient lighting: Ambient lighting is a uniform lighting technique that provides a base level of illumination to the entire scene, regardless of the position or orientation of the objects in the scene.
- Directional lighting: This light simulates a light source that is infinitely far away, providing parallel light rays that illuminate the scene uniformly from one direction.
- Point lighting: Point lighting simulates a light source that is located at a specific point in space and radiates light equally in all directions, creating a realistic sense of illumination and shadow.
- Spotlighting: Spotlighting simulates a light source that is directed towards a specific area, creating a focused cone of light that illuminates objects within the cone and casts shadows outside of it.
- Area lighting: Area lighting simulates a light source that has a finite size and emits light uniformly in all directions, providing soft and even illumination across a surface.
- Image-based Lighting: Image-based lighting (IBL) uses an environment map or panoramic image to simulate the lighting of a scene, providing accurate reflections and lighting that match the surrounding environment.

These lighting techniques can be combined and adjusted in various ways to create realistic and visually appealing scenes in computer graphics. The first four types are shown in the Fig. 1.4.

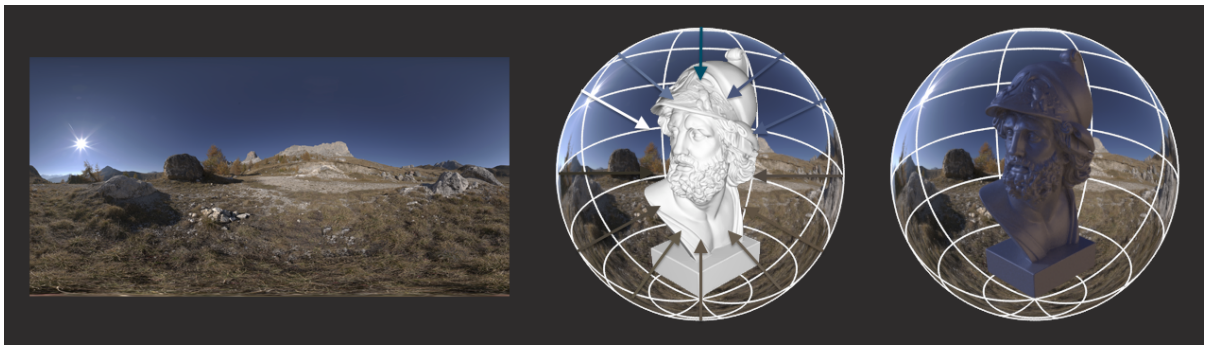


Figure 1.5: Image based Lighting (image courtesy 3delightcloud)

For this thesis, we deal with the last variant of lighting, which is Image-based Lighting. Fig. 1.5. The environment map shown on the left acts as a light source to illuminate the bust in the middle to result in the rendering as shown on the right.

1.4 Materials

Real-world materials are complex to model; most rendering algorithm only approximates their properties. To model them, Bi-directional Reflectance Distribution functions(BRDFs) are used. BRDF is a

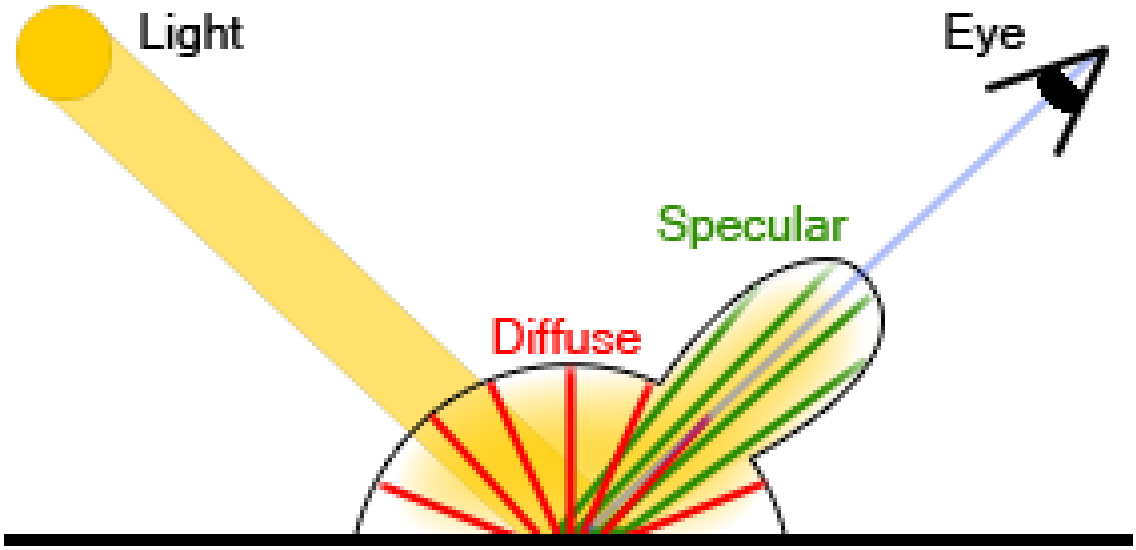


Figure 1.6: BRDF: Diffuse and Specular Response (image courtesy Alto Blogs)

mathematical model that describes the way light is reflected off a surface in a specific direction. It is often used to model the appearance of materials in computer graphics, where it is used to calculate the color of a pixel based on the lighting conditions and the properties of the surface material.

The BRDF function takes as input the incoming light direction, the outgoing light direction, and the surface normal and outputs a value that represents the reflectivity of the surface in that direction. Different materials have different BRDF functions, which determine their appearance and how they interact with light. For example, a glossy or reflective material such as polished metal will have a BRDF function that reflects light mostly in the direction of the specular reflection, while a rough or diffuse material such as a piece of paper will have a BRDF function that scatters light in all directions.

1.5 Appearance Modelling

Now that we have discussed scene representations, lighting conditions, and the material properties, we will discuss briefly how these components are combined together to form images that are either *photorealistic* or *"near"-photorealistic*.

1.5.1 Modelling Light Transport with Monte-carlo Ray Tracing

As discussed earlier, the use of path tracing will help obtain realism to the renderings. It is a form of Monte-Carlo ray tracing that simulates the behavior of light as it bounces off objects in a scene and

interacts with the camera. When a single bounce is modeled, it is referred to as direct illumination, while on the other hand, when multiple bounces produce global illumination.

In path tracing, rays are traced from the camera into the scene, and each ray may bounce off one or more surfaces before reaching a light source or being terminated. At each bounce, the path tracing algorithm calculates the contribution of the light to the pixel being rendered, taking into account the surface properties, such as the BRDF, which describes how the surface reflects light, and the surface normals. The process is repeated for many rays, and the contributions are averaged to create the final pixel color. Path tracing can produce highly realistic images that accurately simulate the effects of indirect lighting, reflections, and refractions.

However, the computational cost of Path Tracing techniques is very high. Techniques such as importance sampling and Russian roulette can be used to reduce computational costs and improve the convergence of the path-tracing algorithm.

The light transport is governed by the Eq. (1.1) proposed by [21, 23]

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_\Omega f_r(x, \omega_o, \omega_i) L(x^{'}, -\omega_i) |cos\theta_i| d\omega_i, \tag{1.1}$$

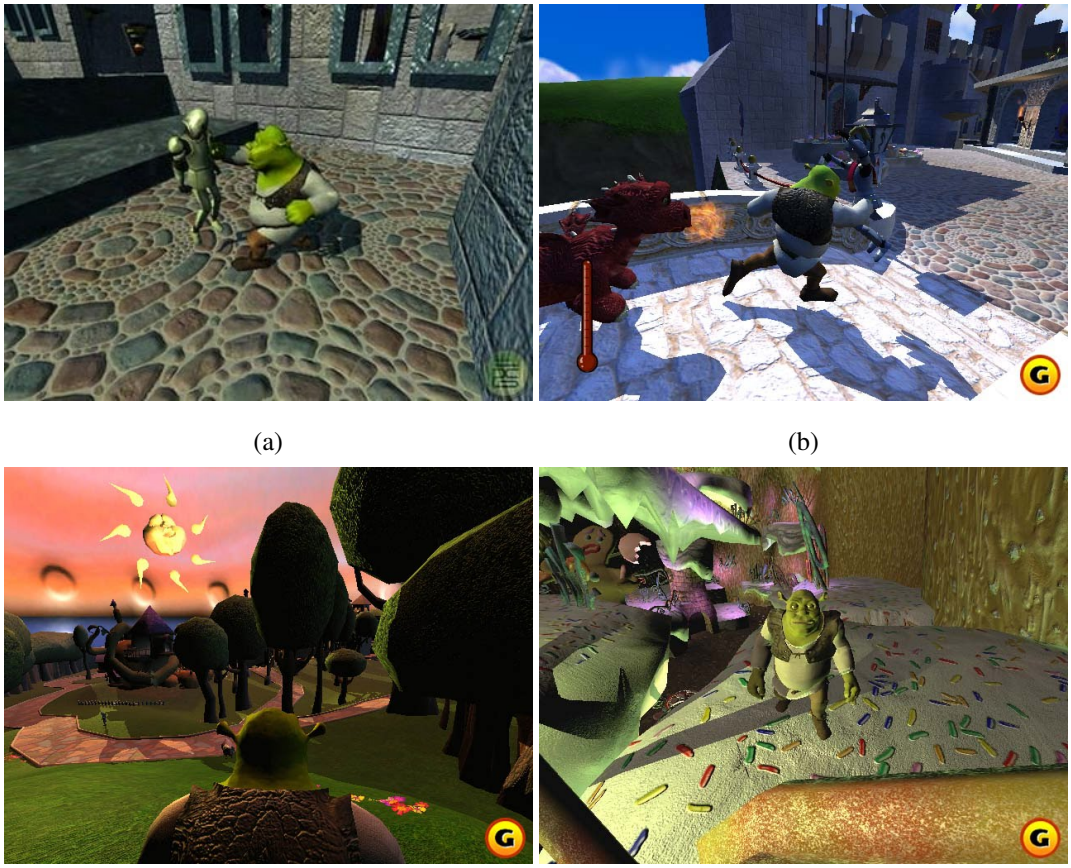
where

- $L_o \rightarrow$ final radiance at a point x as observed from the direction of ω_o
- $L_e \rightarrow$ emitted radiance from point x towards ω_o.
- $f \rightarrow$ BRDF
- L $\rightarrow$ incoming radiance towards x from $x^{'}$ in the direction of $-\omega_i$
- $cos\theta \rightarrow$ cosine of the angle between the incoming direction $-\omega_i$ and normal at $x^{'}$.
- $\Omega \rightarrow$ imaginary unit hemisphere around the point x.

The rendering equation is a complex integral equation that cannot be solved analytically for most scenes, and numerical methods such as Monte Carlo integration are typically used to approximate the solution. The Eq. (1.2) shows the approximate formulation of Eq. (1.1) where N independent samples of ω_j are sampled from a distribution which has a probability of $p(\omega_j)$. The quality of the estimate is dependent on the N as well as the distribution function. A higher sample rate will usually result in converged and more accurate results. It should also be noted that a wrong distribution from which sampling is made will adversely affect the obtained result; often, these results are termed biased results.

The use of (multiple) importance sampling strategies improve converges rate times provided a sufficient understanding of the material and/or lighting model is available.

$$L_o(x, \omega_o) \approx L_e(x, \omega_o) + \frac{1}{N} \sum_{j=1}^{N} \frac{f(x, \omega_o, \omega_j) L(x^{'}, -\omega_j) |cos\theta_j|}{p(\omega_j)} \tag{1.2}$$



(a) (b)

(c) (d)

Figure 1.7: Deferred Lighting (image courtesy Shrek 2001)

1.5.2 Deferred Rendering

Due to the computational complexity of Monte-Carlo-based ray tracing, methods like Deferred Rendering/Shading have been employed. These methods provide multi-pass geometric buffers, often referred to as G-buffers like position, normal, view direction, lighting, etc., and utilize a simple Blin-phong model [3] to approximate the photorealism. The first video game to utilize this technique was SHREK[1] in 2001. As can be observed, the lighting of the game changes dynamically. Most of the effects seen here are shadows either shadow mapped or obtained by multi-pass renders. It is not physically based, as can be observed in Fig. 1.7d with the shadows and lighting being mimicked using multi-pass rendering strategies.

While deferred rendering is initially used to approximate the effects of ray tracing, the performance benefits it provides by eliminating the redundant fragments are substantial. In a typically OpenGL-based rendering pass, there are quite a few fragments. Specifically in scenes with a high depth complexity

[1] https://web.archive.org/web/20131202224623/http://www.electricsheepgames.com/games3

(multiple objects cover the same screen pixel), the same pixel region will fire multiple fragment shader outputs and is overwritten once by the nearest point based on the Z-buffering information.

Hence, these aspects of deferred shading are still utilized in the industry to eliminate the excess fragments and only process the visible geometry in both path-tracing and rasterization frameworks.

1.5.3 Analytical approximation of Path Tracing effects

Another interesting direction to obtain the effects of global illumination is by using analytical approximations. Recent work on Linearly transformed cosines, which provide a closed-form solution to the light contribution of an area light [17].

Earlier methods like Efficient representation of Irradiance Environment Maps [47] by Ramamoorthi et al. provide the foundation for distant illumination models. The work proposes to use Spherical Harmonic(SH) representation to define environmental irradiance. The work does not account for soft shadows nor renders images in real time.

Subsequent work by Sloan et al. [53] provided a framework for evaluating the final radiance at every point on the scene while also being real-time. It does this by disentangling the components of the rendering equation.

$$L^p(\omega_o) = \int_\Omega L(\omega_i)\rho^p(\omega_o, \omega_i)V^p(\omega_i)(\omega_i \odot n)d\omega_i \tag{1.3}$$

where ω_o is viewer direction from p, ω_i is incoming direction on unit hemisphere Ω and n is the surface normal at p. L^p is the reflected radiance in direction ω_o, L is the incoming environment light from ω_i, V^p is the binary visibility function and ρ^p is the Bi-directional Reflectance Distribution Function (BRDF).

The Eq. (1.4) is decomposed into distant lighting and transfer signals as shown below:

$$L^p(\omega_o) = \int_\Omega L(\omega_i)\rho^p(\omega_o, \omega_i)V^p(\omega_i)(\omega_i \odot n)d\omega_i \tag{1.4}$$

The distant lighting $L(\omega_i)$ and transfer signal $V^p(\omega_i)(\omega_i \odot n)$ are precomputed individually, and projected to Spherical Harmonic basis [47].

The calculation of lighting is done by projecting every point on the environment map into spherical harmonics bases. On the other hand, the Transfer signal is calculated for every geometric point on the surface of a geometry based on cosine sampling. This is done by shooting numerous rays from every geometric point, which provides information about visibility in the direction of the ray. Each such visibility is projected into a spherical harmonic basis, and finally, an effective transfer signal at the point accounting for the visibility is obtained in the form of an SH vector.

While the Spherical harmonic-based integration provides a fast rendering framework for producing real-time soft shadows and dynamic illumination changes, it is constrained by the representative capacity, constraining it to low-frequency illumination changes and distant lighting, and static scenarios. Despite these disadvantages, the setting is highly suitable for scene settings where objects are static, for

example, buildings in the scenes. Hence we choose this fast analytical framework for the exploration of the real-time rendering scenarios for arbitrary geometric representations.

1.6 Contributions

For this work, we have chosen Spherical Harmonics based Precomputed Radiance Transfer for the rendering of arbitrary surface geometries. The *Lighting* is calculated per environment map and only comprises a memory of a vector or a single matrix. On the other hand, *Transfer* signal needs to be calculated and stored for every point on the surface geometry in an ideal scenario. But sampling such a dense surface is computationally challenging. Hence, prior art like [53] has utilized vertex space for the storage of Transfer signals.

While usage of densely tessellated meshes results in artifact-free render, lowly sampled/tessellated meshes suffer greatly to render the desired photorealistic effects. Secondly, when the representation of the scene is not mesh based, like Implicit surface *(SDF)*, the discrete storage of such a Transfer signal is infeasible. This thesis tackles both problems and addresses them while maintaining real-time framerates and providing artifact-free renders.

Transfer Textures for Fast Precomputed Radiance Transfer

In the case of densely tessellated meshes, the barycentric interpolation of the Transfer signal usually produces faithful results. But in the case of simple geometries (e.g. Walls, Planes, etc) that do not require high tessellation for accounting for the geometric details, it is often redundant. Hence we propose the usage of area-preserved UV-map storage of Transfer Signal. This helps produce artifact-free renders as well as maintain lower memory footprints and computational complexities.

Real-time Rendering of Arbitrary geometries using Learnt Transfer

The absence of the inherent storage schema (1) Vertex Mesh and 2) UV mapping) for storing *transfer* signals limit the extension of the PRT framework to implicit surfaces like SDFs. We propose the use of the Neural Surface Function Approximators [19] to provide a surface mapping from surface position to Transfer Signal. Major challenges of maintaining faster framerates are handled by checking various aspects like Fragment Shader capacities for handling Neural Weights and providing a generic framework employing CUDA.

In both works, the Transfer signal is clubbed with desired Lighting and BRDF using [39] to obtain final radiance at every point. Rendering the whole array of fragments produces the final renders of the scene.

1.7 Thesis Layout

In this thesis, we divide the content mainly into 5 Chapters. The Chapter 1 discusses briefly the problem setting of PRT, the geometric constraints, and some relevant literature to give a broad overview.

In Chapter 2, we provide a study of prior arts existing in the field of PRT for real-time rendering and solutions proposed by various works. Finally, we contrast the prior art with our approach.

In Chapter 3, we present the area-preserved UV-mapped Transfer storage framework for PRT. We also show qualitative results against the SOTA and finally report the performance metrics both in render times and memory footprints.

In Chapter 4, we present a framework that is robust to the geometric representation and can work in real-time framerates while providing artifact-free renders.

Finally, Chapter 5 concludes the thesis with remarks and future directions.

Chapter 2

Related Works and Background

In this chapter, we will discuss the relevant literature for the problem addressed in the thesis. We first discuss the existing literature on Precomputed Radiance Transfer. We then discuss their usages in different scene representational modalities. Later we mention the methods which can be employed for arbitrary geometric representations. Additionally, we also discuss the relevant literature inspired by which we build our solution for the two problem settings of Low-poly meshes and arbitrary geometries. We finally contrast against the relevant work and provide the required mathematical background for explaining the integration of the Rendering equation using the suitable band-limited approximation of Spherical Harmonics.

2.1 Related Work

Spherical Harmonic Representation: Spherical Harmonic (SH) lighting was first proposed by [47]. The work focused mainly on representing the environmental lighting in the form of Spherical Harmonics basis. This representation was aimed to handle low-frequency distant lighting scenarios efficiently. The lower orders of Spherical Harmonic basis were employed in cases of diffuse material lobes while the specular lobes demanded a high-order SH basis.

Traditional PRT: This compact and efficient representation of irradiance maps has led to the development of PRT, proposed by [52, 53] to disentangle rendering equations into transfer, lighting, and material while individually projecting them to the SH domain. Since then, PRT and SH have received a lot of attention to efficiently compute SH basis [55], efficient rotation of SH [40], compressing SH basis [54], microfacet BRDFs [26] and extending PRT for dynamic scenes [69]. But the fundamental idea of disentangling and relevant mathematical formulation has mostly been unaltered.

For diffuse materials, the transfer signal is stored as a vector while for glossy materials the transfer signal was stored as a matrix. The matrix representation is used to handle the interaction between the change of viewing directions. While the matrix representation can faithfully replicate the glossy effects, the storage of the matrix for every point on the surface was substantially demanding. This problem was

addressed by the work of *Triple Product Formulation* [39] which allows storage of a vector for transfer even in-case of glossy materials by using the tripling coefficient matrix.

Triple Product Formulation: Triple products naturally arise in computer graphics in the rendering equation. Triple products in wavelet basis and spherical harmonics have been studied in depth [39]. Today, state-of-the-art in PRT uses triple products for dynamic relighting for diffuse and glossy scenes. By itself, the triple product method has a computational complexity of $O(k^3)$. This method can be made more computationally efficient by fixing the lighting [52]. Specifically, triple products with fixed lighting in SH-based PRT achieve a computational complexity of $O(k^2)$ and per-vertex storage of k-dimensional vector. We augment the triple product method with transfer textures and demonstrate superior rendering quality and real-time framerates.

As the problem of reducing transfer storage is addressed by the triple product formulation, the storage of transfer vectors has been handled majorly in two spaces. 1) Vertex Space 2) Texture Space

Vertex Space: Most traditional approaches [26, 41, 46] as well as the newer ones for like polygonal lighting shading in PRT [2, 63, 64], fast spherical harmonic product [65] store transfer at vertices. For producing artifact-free renders all of these approaches necessitate dense tessellation to account for transfer changes at high frequency in the scene. To handle these issues adaptive re-meshing techniques were proposed by [25] which store transfer at vertices while re-meshing the region where the shadows are missing. (refer Fig. 2.1)

Texture Space: Works by [22, 34] store low-order SH coefficients at UV-mapped textures to account for diffuse results confining to the *diffuse* material models. Additionally, they are also constrained to the direct illumination model, disregarding the indirect illuminance.

Meshless geometries: The limitation of these methods is they are constrained to mesh representation either by using a vertex storage approach or by using a texture storage approach.

FEM approaches To remove the dependency on the mesh representation, Meshless hierarchical transport proposed by [28] samples and stores selective points by accounting for high-fidelity changes in the transport function. But this approach requires multiple re-sampling and weighted k-nearest neighbor searches to obtain a transfer to a given query point. All of these methods rely on a discrete representation of transfer.

Irradiance Volumes Another approach to disregard the inherent storage structure present in the mesh is by using a volumetric grid of spheres providing spatial storage of spherical harmonic transfer coefficients [14]. This method has widely been used in various works like games [18,59], glossy reprojections [50], compressed representation of precomputations [51], Chrominance [61], and interactive lightmaps for frostbite BRDF [1]. Recent advances have utilized the representation to obtain dynamic diffuse global illumination [32], with efficient Sign distance improvements over [20], while also extending to

Neural Light probes [15]. Despite these efforts, the Irradiance Volume representation is prone to light leaks causing artifacts as shown in Fig. 2.2.

Later and recent efforts like [33] have provided solutions to eliminate light leakages. Nevertheless, the use of irradiance volumes requires high memory storage in the orders of N^3. Additionally, the latest line of works [32] uses UV-mapped spherical probes which hold precomputed values in texture space. Maintaining N^3 textured precomputations is usually very memory intensive.

Neural Precomputed Radiance Transfer (Neural PRT) Nascent efforts like NeuralPRT by [45] and Neural Radiance Transfer Fields by [30] take inspiration from PRT, learning latent representation for transfer with lighting, diffuse and glossy descriptors. Furthermore, they operate in the image space and perform loss calculations in the image domain, whereas we operate in the SH space. For disentangling components from the final rendered images, they employ large neural MLPs for their constituent rendering components, limiting their run times. We on the other hand stick to traditional formulations of PRT allowing our method to be easily integrated into existing frameworks like [63, 64].

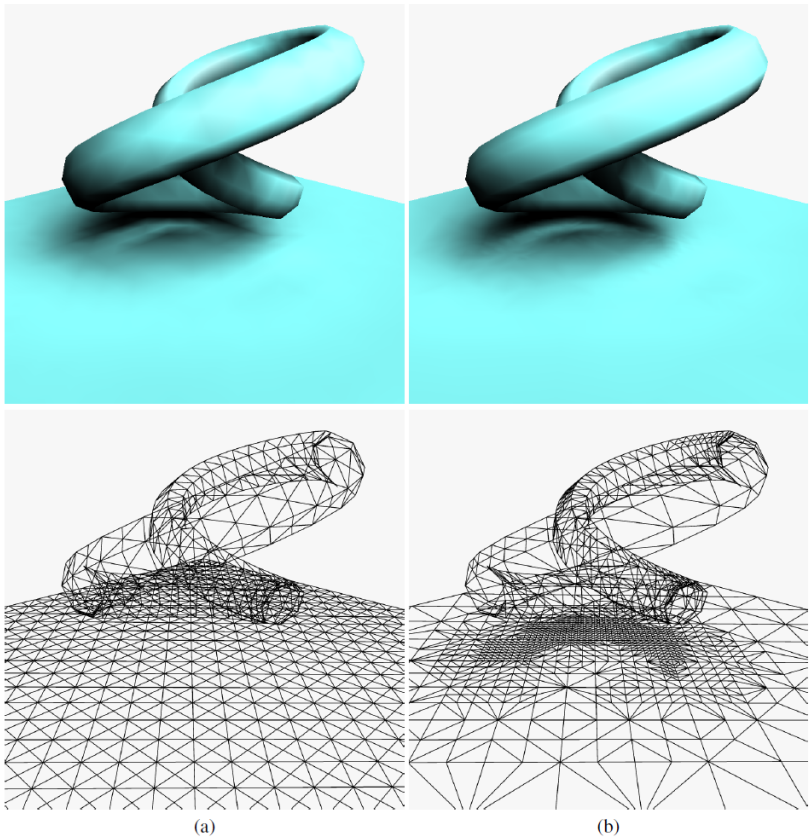


Figure 2.1: Adaptive Tessellation of Geometry to accommodate accurate shadowing effects. Observe the higher tessellation at the regions under the manifold which alleviates the requirement of dense mesh structure on the ground plane. (image courtesy [25])

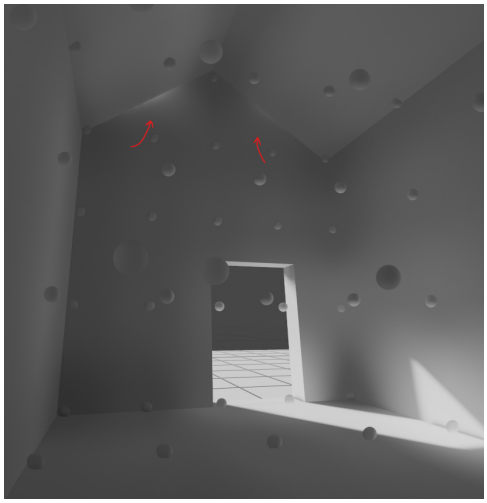


Figure 2.2: Use of probe also accompanies Light leakage, causing artifacts as shown in the figure. Irradiance Volume usually requires careful placement of Light Probes. (image courtesy Blender artist forums)

Neural Fields In recent times the use of neural fields, which learn a function representation to augment the representation of scene parameters has shown promising results. The work of NeRF [35] is one such classical example. The use of simple MLPs to approximate the radiance of the scene has improved the novel view generation from a given input set of discrete images. While some follow-up works like [16, 48] concentrated on making the rendering real-time, some works looked at the semantics of the Radiance Fields [4, 68]. There have also been works in the direction of editing materials and lighting [5, 38, 67] and geometry by [6, 66]. Contrary to their efforts our work concentrates mainly on real-time rendering along with the facilitation of material and lighting edits.

The use of simple MLPs to represent functions like radiance or irradiance has been proposed earlier in the works of [49]. While the works like [49] try to regress radiance, we aim to regress the transfer. Recent efforts like [29] have also used MLPs to regress transfer. But the primary concentration of the work was to handle deformations in the geometry while utilizing harmonic maps of the temporal changes.

Contrary to efforts of DeepPRT [29] which use neural approximations of transfer, we do not require UV mapping and our method can pan to arbitrary surface representations. In contrast to works of NeuralPRT [30,45] which re-model PRT formulation and work in latent space, we stick to the traditional formulation making our method extendable to works like [63, 64]. On the other hand, we differentiate over the NeRF [35] greatly as we do not fit a model to understand the scene from a discrete set of images, rather we regress the irradiance represented in SH while maintaining real-time framerates. Though works like KiloNeRF [16,48] provide real-time renders they do not facilitate material and lighting edits. NeRFactor [67] which extends NeRF to facilitate editing lacks real-time rendering.

2.2 Background

The work of Sloan et al. [53] was introduced to obtain ray-tracing effects in real time. The method disentangles *lighting* and *cosine-weighted visibility* into two different parts. The cosine weighted visibility is coined as *transfer* [53]. The separated entities are precomputed and stored with a change of basis. As all the sub-functions of the rendering equation are spherical, a suitable spherical domain is chosen. Taking inspiration from Sloan et al. we also chose to use Spherical Harmonic representation(SH basis).

The rendering equation for direct lighting at point p is given by

$$B^p(\omega_o) = \int_\Omega L(\omega_i)\rho^p(\omega_o, \omega_i)V^p(\omega_i)(\omega_i \odot n)d\omega_i, \tag{2.1}$$

where ω_o is the direction towards the viewer from p, ω_i is the incoming direction on the unit hemisphere Ω and n is the surface normal at p. B^p is the reflected radiance in direction ω_o, L is the incoming environment light from ω_i, V^p is the binary visibility function and ρ^p is the Phong Bi-directional Reflectance Distribution Function (BRDF) [3]. Eq. 2.1 is decomposed into the lighting L and transfer $T^p(\omega_i) = V^p(\omega_i)(\omega_i \odot n)$ which are then projected to the SH basis with coefficients $\mathcal{L}_i$ and $\mathcal{T}_i^p$ respectively. The term $\mathcal{T}_i^p$ is referred to as *transfer*.

The original work Sloan et al. has two different representations of *transfer* for diffuse and glossy materials. In the case of *diffuse* materials, the $\mathcal{T}_i^p$ is a vector of k-dimension to represent an SH-basis of transfer. In case of *glossy* the $\mathcal{T}_i^p$ is a matrix of size k^2. The size of the transfer representation increased by k times. This has been addressed with the help of Triple product formulation [39]. Using [39] it is only required to store a k-dimensional vector for both glossy and diffuse materials. This paves way for our idea of learning the transfer vector using a network Chapter 4.

The diffuse case is fairly simple the radiance B^p can be directly calculated as:

$$B^p = \sum_i \mathcal{T}_i^p \mathcal{L}_i. \tag{2.2}$$

.

For the calculation of radiance for a glossy surface material, we have to evaluate the triple product formula [39].

$$\mathcal{H}_k^p = \int_{S^2} y_k(\omega) \left(\sum_{i=1}^{n^2} \mathcal{T}_i^p y_i(\omega) \right) \left(\sum_{j=1}^{n^2} \mathcal{L}_j y_j(\omega) \right) d\omega = \sum_{ij} \tau_{ijk} \mathcal{T}_i^p \mathcal{L}_j, \tag{2.3}$$

The terms $\mathcal{L}_j$ and $\mathcal{T}_i^p$in Eq. 2.3 account for *lighting* and *transfer* respectively. Hence obtained $\mathcal{H}_k^p$ is convolved with BRDF f^p to obtain the final radiance B^p.

$$B^p = \mathcal{H}_k^p \odot f_k^p \tag{2.4}$$

The τ_{ijk} term here is a tripling-product matrix which is a $3D$-matrix of dimension $i \times j \times k$. It can be obtained using the triple-project of basis functions described in [39]. In most works, this tripling matrix is of the same dimensions across all three axes of the $3D$ matrix. (i.e; $i = j = k$).

But the formulation by itself does not constrain the user to use the $i = j = k$. More elaborately, we $\mathcal{T}_i^p$, can be a vector of size i, $\mathcal{L}_j$ can have dimensionality of j while maintaining $(i, j) \neq k$. This enables flexible detailing of parts of the rendering equation which contain high-frequency details. For example, BRDF is usually a high-frequency function that requires more coefficients to represent. In such cases the dimension k in τ_{ijk} can be increased without re-computing the transfer or lighting. This is consistent with works like [63, 64].

As we only learn *transfer* in our case, our approach can also leverage and benefit using the Triple Product Formulation [39].

2.2.1 Spherical Harmonics

Spherical Harmonics (SH) are orthonormal basis functions on the unit sphere, parameterized by a direction $\omega = (\theta, \phi)$, the band l and order m. The real spherical harmonics are given by:

$$Y_m^l(\omega) = \sqrt{\frac{(2l+1)}{4\pi}\frac{(l-|m|)!}{(l+m)!}} P_{|m|}^l(\cos\theta) f(|m|\phi), \tag{2.5}$$

where $P_{|m|}^l$ are the associated Legendre Polynomials. The function $f(|m|\phi)$ is 1 when $m = 0$, and is equal to $\sqrt{2}\cos(m\phi)$ when $m > 0$ and is equal to $\sqrt{2}\sin(|m|\phi)$ when $m < 0$. Projecting a function to band l SH basis results in a vector of l^2 coefficients, from which the original function can be recovered by summing over all SH bases [46]. Often, it is convenient to index SH coefficients with a single index $i = l(l+1) + m + 1$ ranging from 1 to n^2.

2.3 Contrast

In the early efforts of PRT [53], the *transfer* was stored in vertex spaces, which required either a dense tessellations or adaptive re-meshing [25]. To avoid this, we draw inspiration from the *textured storage* methods to store transfer functions in UV-mapped spaces. These UV-mapped spaces are more continuous, providing artifact-free renders. Unlike prior work of [34] restricted to diffuse renders, we adopt triple product formulation extending the method to glossy materials. We further extend their work by incorporating glossy materials and inter-reflections in the Textured PRT frameworks. We discuss this in detail in the Chapter 3.

In the scenarios of rendering meshless-geometric representation, which lacks an inherent storage schema, we propose to use neural approximators. This eliminates the high memory requirements caused by techniques like Irradiance Volumes [14] while providing a continuous representation of transfer. This approach also avoids dealing with light leaks. We discuss this in detail in the Chapter 4.

Though we utilize neural approximators for *transfer* regression, we do not follow similar approaches as NeuralPRT [30, 45]. NeuralPRT proposes using latent representations for all lighting, material, and transfer signals, invalidating the use of Triple-product formulation. Unlike them, we leverage traditional

Triple-product-based evaluation to extend current SH-based pipelines to incorporate arbitrary surface geometries.

Chapter 3

Transfer Textures for Fast Precomputed Radiance Transfer

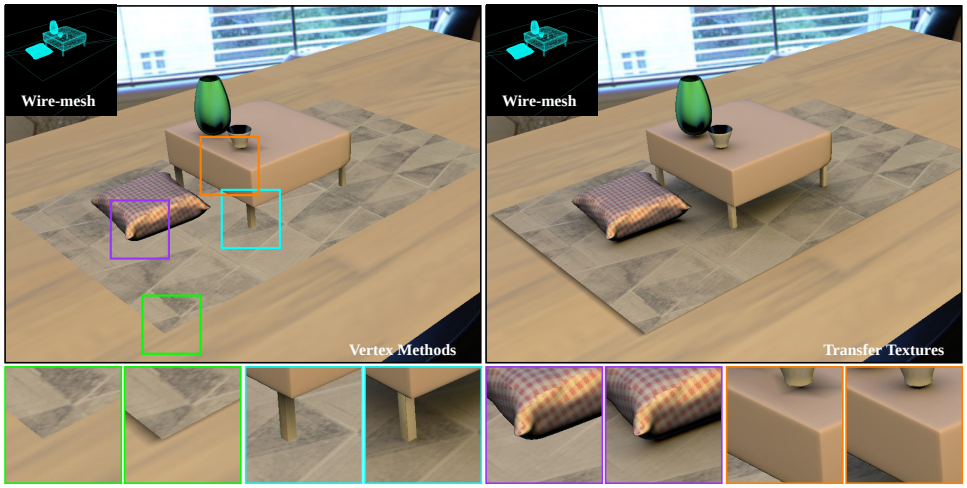


Figure 3.1: The above scene shows a minimally tessellated rug, floor, and table (shown in wireframe insets). The vertex attribute method (left) fails to capture shadows due to insufficient sampling of the transfer while the use of our transfer textures accurately captures all details for the same tessellation.

In this chapter, we present a storage strategy to alleviate the issue of using dense tessellation for the work of PRT [53]. We present transfer textures to decouple mesh resolution from transfer storage and sampling. We do this by utilizing the area-preserving UV spaces of the scenes creating *Transfer Textures*. Transfer textures are UV-mapped textures storing sampled *transfer* function at every scene location. The relatively more continuous nature of textures over the regular vertex attributes provides artifact-free renders. This alleviates the necessity of redundant tessellation. Our method evaluates final radiance in a fragment shader via sampling the *Transfer-Texture* unlike vertex color interpolation of the prior art of [53]. Additionally, we also provide a method to compute 1-bounce indirect illumination to obtain indirect illumination by fixing the distant lighting. Our method achieves real-time rendering framerates while obtaining artifact-free renders.

3.1 Transfer Textures

In this section, we begin with a description of computing and storing a band l SH projection of transfer on a texture (Sec. 3.1.1). Next, we show how inter-reflections can be pre-computed and incorporated into our framework (Sec. 3.1.2). Our implementation is described in Sec. 3.2. Our approach achieves real-time frame rates and better render quality, especially on low tessellation meshes, as shown in Sec Sec. 3.3.

Algorithm 1: Pre-computing and storing the transfer texture.

Input: $\mathcal{M}, w, h, l$: Mesh $\mathcal{M}$, width w & height h, SH band l.

Output: T_o: Precomputed transfer texture

1 $T_o \leftarrow$ Texture(w, h, l) `// Init. texture.`

2 $\mathcal{G}$ = GenerateGBuffer($\mathcal{M}$) `// G-Buffer`

3 **for** *t in* T_o **do**

4 point = $\mathcal{G}$[t.x][t.y].vertex

5 normal = $\mathcal{G}$[t.x][t.y].normal

6 V = ComputeTransfer(point, normal) `// Path tracing`

7 V_{sh} = SHProject(V)

8 T_o[t.x][t.y] = V_{sh}

9 T_o = Dilate(T_o, 3)

3.1.1 Pre-computing Transfer Textures

The computation of transfer involves shooting multiple rays from a point p in the scene and then evaluating and projecting the transfer to the SH basis. For transfer textures, there are N scene points p corresponding to each pixel t in the texture. The mapping between t and p is defined by the UV coordinates. To efficiently compute the transfer texture, we leverage G-Buffers(Alg. 2) to interpolate vertex positions and normals based on their corresponding UV-Coordinates (Alg. 1, line 2). Next, we read the G-buffer and the scene geometry and evaluate the transfer function for each pixel in the buffer (Alg. 1, lines 4-6). The transfer obtained is then projected to SH basis and stored at the same pixel location in an initially empty texture T_o (Alg. 1, lines 7-8). Finally, T_0 is dilated to ensure that all points inside a triangle receive a transfer value. At run-time, we fetch transfer T_o and use it with the triple product formulation to obtain B^p.

Algorithm 2: Generate G-Buffer pass.(Vertex and fragment code)

```
1  Program VertexShader:
2      vec4 gl_Position;                                   // In-built variable
3      in vec3 p, n;                                       // Scene Point, Normal
4      in vec2 uv;                                         // UV co-ordinates
5      out vec3 vertex;                                    // Interpolated in Frag.
6      out vec3 normal;                                    // Interpolated in Frag.
7      void main():
8          gl_Position = vec4(uv.x, uv.y, 0.0, 1.0);
9          vertex = p; normal = n;
10 Program FragmentShader:
11     in vec3 vertex;
12     in vec3 normal;
13     out vec4 gPos;                                      // G-Buffer
14     out vec4 gNorm;                                     // G-Buffer
15     void main():
16         gPos = vec4(vertex,1.0);
17         gNorm = vec4(normal, 1.0); /* w → alpha channel                */
```

3.1.2 Pre-computing transfer textures for inter-reflections

Inter-reflected radiance B_i^p at point p can be modeled as:

$$B_i^p(\omega_o) = \int_\Omega (1 - V^p(\omega_i)) B^{pq}(x, \omega_i) \rho^p(\omega_o, \omega_i)(\omega_i \odot n) d\omega_i, \tag{3.1}$$

where B^{pq} is the radiance from a secondary hit-point q towards p and B_i^p is the inter-reflected radiance [53]. First, we factor out $1 - V^p(x, \omega_i)$ by only integrating over rays that hit some geometry. For a scene point p_1 and a secondary hit q_1, the radiance $B^{p_1 q_1}$ can easily be precomputed given a zero-bounce transfer texture T_o from Alg. 1 (See Fig. 3.2). The radiance from q_1 towards p_1 is obtained

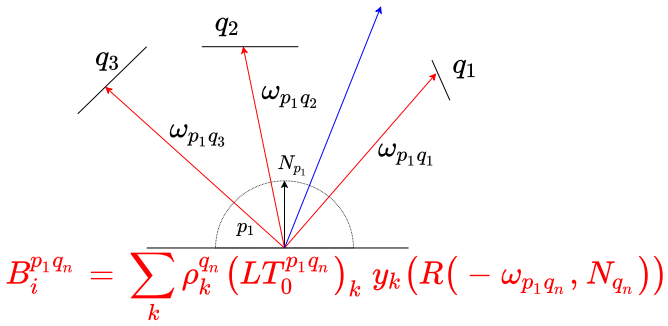


Figure 3.2: **Inter-reflections:** The radiance $B^{p_1 q_n}$ (red lines) towards a point p_1 from a secondary hit-point q_n can be computed by first fetching transfer at q_n using the zero-bounce transfer texture T_o, applying the light L followed by convolution with the BRDF at q_n and evaluation at reflected direction along the normal at q_n. $B^{p_1 q_n}$ forms an indirect environment map which is projected to SH and stored at p_1 in an additional texture.

using the triple product formulation by fetching T_o to obtain transfer at q_1.

This is done for all hit points from p_1. This radiance now forms an *indirect environment map* for the point p_1, which is then projected to SH basis resulting in a k-vector B_i^{pq}, which is stored in a separate *one-bounce inter-reflection* texture T_1. At run-time, the inter-reflected radiance is obtained by convolving B_i^{pq} fetched from T_1 with the BRDF SH ρ_i^p and evaluating at the reflection direction. The final color is given as: $B^p(\omega_o) + B_i^p(\omega_o)$. Alg. 1 can be easily extended to compute the second bounce texture T_2 and so on. The number of textures required is linear in the number of bounces in this setting, and the final color is just their summation.

3.1.3 Handling dense UV-packing

In the previous section, we described methods for the efficient computation of transfer textures. Usage of these textures requires UV co-ordinates each vertex to be defined. To obtain UV unwrapping

of scene geometry, we used *Smart UV-Unwrap* or *Light Map Pack* from Blender 3D [7]. One caveat with UV unwrapping is that dense packing of UV islands may cause overlaps which manifest as rendering artifacts. *Smart UV-Unwrap* does not guarantee non-overlapping islands while *Light Map Pack* leads to texture wastage and tiny pixel coverage for some parts of the geometry. In such scenarios, texture-sets are beneficial.

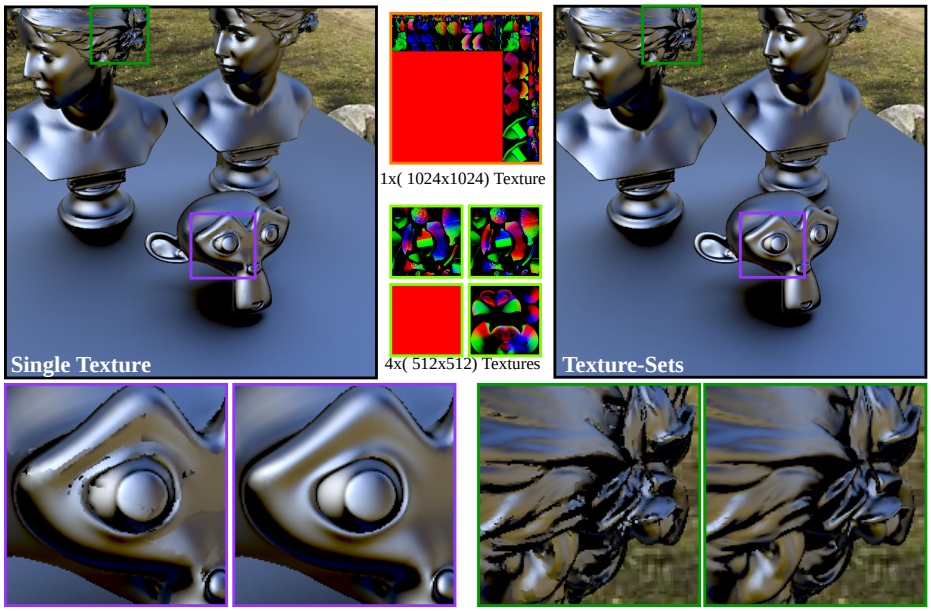


Figure 3.3: **Texture sets:** TRM:Two Roza, one Monkey, We demonstrate the use of texture-sets with TRM(right) where 4 small textures are assigned to each piece of geometry against the use of single texture in case of TRM(left). The artefacts can be seen clearly in the Monkey's eyes and Roza's hair as depicted in insets. Note that in both cases, the memory requirements are the same (single 1024×1024 texture v/s four 512×512 textures).

Consider an example scene as shown in Fig. 3.3. This scene contains 441K triangles, all of which are packed into a single 1024×1024 texture (Fig. 3.3, left). As shown in the insets, this leads to artefacts. A better approach is to use texture-sets, which means assigning individual textures to each object in the scene (Fig. 3.3, right). In this case, each UV island can occupy the entire space of the texture thus eliminating artefacts.

3.2 Implementation Details

We implement Alg. 2 in Python using the ModernGL [11] framework. We generate and store the resulting G-buffers for each scene in a pre-process step. Alg. 1 is implemented in Python and uses Embree [62] for efficient ray intersection tests. We project to band $l = 5$ (25 coefficients) real spherical

harmonics. As mentioned in Sec. 3.1.1 dilation is required to ensure that all points in the scene receive a transfer value. Experimentally, we found a dilation of three to be sufficient which may need adjustment depending on the scene complexity. The time taken for generating transfer textures for a scene like in Fig. 3.1 is approximately three hours.

Our real-time renderer is also implemented in the ModernGL framework. We implement the triple product (TP) and triple product with fixed light (TPFL) methods augmented with our transfer textures. Rendering is done in the fragment shader using the generated transfer textures for the respective scene. We render all scenes with glossy materials with spatially varying roughness on a workstation with an NVIDIA RTX 3090 with a resolution of 1920×1080. An important detail is that we use the early depth pass to prune fragments that are not visible thus avoiding unnecessary computations. We use a texture resolution of 1024×1024 texture as we have found it to be best trade-off in between memory and quality for our scenes.

Table 3.1: Scene configurations. We list all scenes used in this paper, with their corresponding number of triangles and FPS with triple product (TP) and triple product fixed light (TPFL) methods on both vertex and fragment shaders (with transfer textures). Our approach achieves real-time framerates on all scenes.

Scene	**# tris.**	**Vert. (Trad.)**		**Frag. (Ours)**	
		TP	TPFL	TP	TPFL
Dragon (Fig. 3.4)	1.3M	3.62	41.2	5.2	151.2
TRM (Fig. 3.4)	441K	10.2	116.2	15.2	202.9
Room (Fig. 3.1)	21K	352.3	2432.7	83.6	568.2
Plants (Fig. 3.4)	18K	363.2	2597.6	6.7	168.3

3.3 Results & Evaluation

In this section, we present glossy rendering results including inter-reflections using transfer textures on the fragment shader. We compare the renderings with traditional vertex shader based approaches. We also discuss and demonstrate the use of normal maps with transfer textures which is not possible with traditional vertex based PRT. Finally, we analyze the memory requirements and give a lower bound of FPS for tranfer texture usage in a fragment shader. Rendering results are demonstrated on four scenes whose statistics and performance comparisons are given in Tab. 3.1.

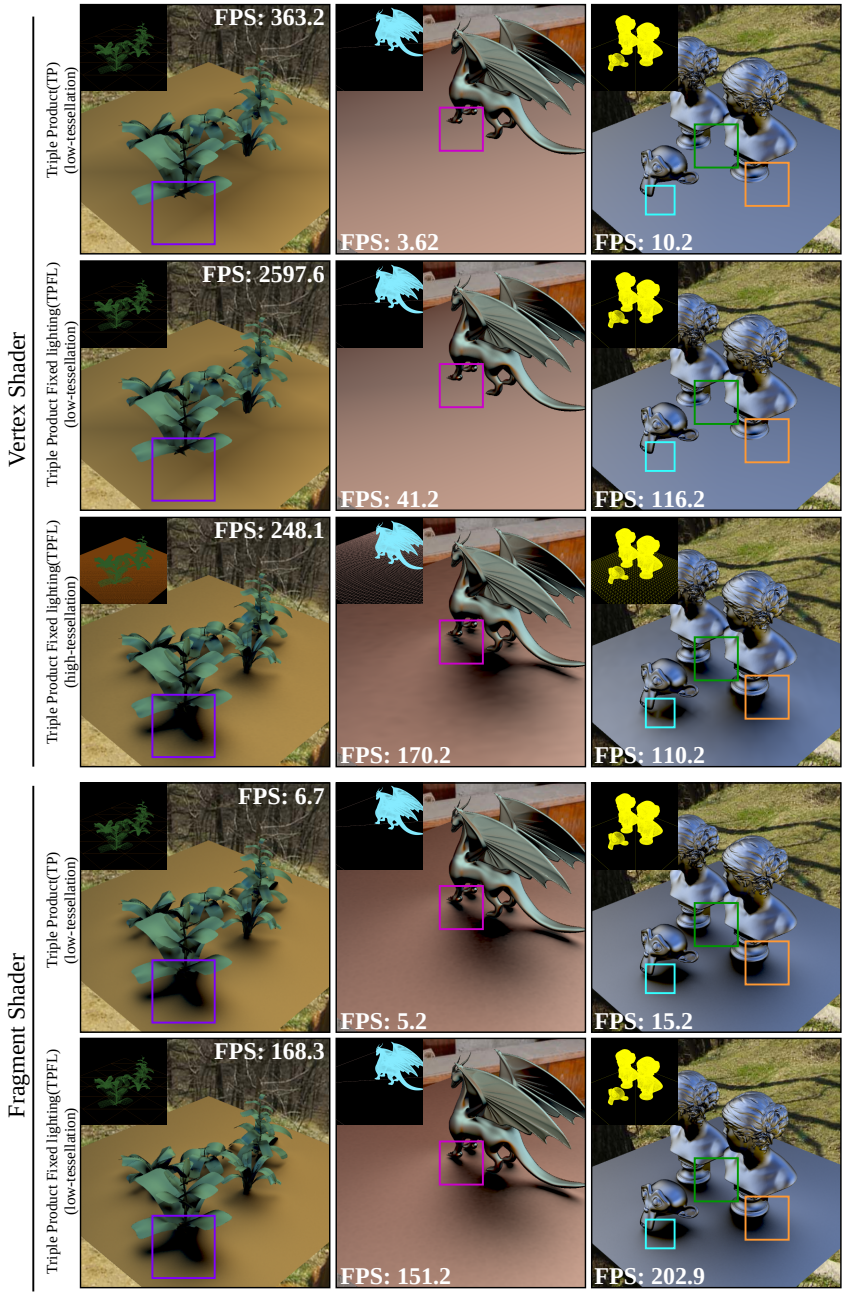


Figure 3.4: We show results of TP and TPFL on the fragment shader using our transfer textures (Bottom). We compare renderings with traditional vertex shader based approaches on the top. For minimal tessellations, our method accurately renders shadows whereas previous methods are unable due to insufficient sampling of transfer. The third row (high-tessellation) shows that renderings using traditional methods approximately approach the quality of transfer textures on addition of more vertices. Note that low-FPS in case of Dragon low-tessellation and TRM low-tessellation in vertex based TP and TPFL is due to their high resolution geometry.

3.3.1 Glossy rendering & Inter-reflections

Fig. 3.4 shows the renders for three scenes: *Plants*, *Dragon* and *TRM (two Roza, one Monkey)*. All scenes have a ground plane, which is minimally tessellated, as shown in the wireframe insets. The TP and TPFL methods on vertex shader are unable to capture proper shadows on the ground plane due to sparse sampling of the transfer function. In contrast, the TP method on the fragment shader using our transfer textures properly reproduces shadows on the plane, albeit at a very low FPS. The TPFL method with transfer textures also achieves a similar render quality at a higher FPS. We note that the TP/TPFL methods on vertex shader approach the render quality of our transfer textures with a highly tessellated ground plane, as shown in the *high-tessellation* renderings. We note that this requires the addition of *redundant* vertices. We further note that such situations frequently arise in production, for example with walls in a room or any large surface with minimal curvature (Fig. 3.1). In such cases, all previous PRT methods on vertex shaders require the addition of avoidable vertices to store the transfer on leading to drop in performance, as opposed to our transfer textures method. Additional renders with different phong exponents and environments maps for four different scenes are shown in Fig. 3.7 & Fig. 3.8.

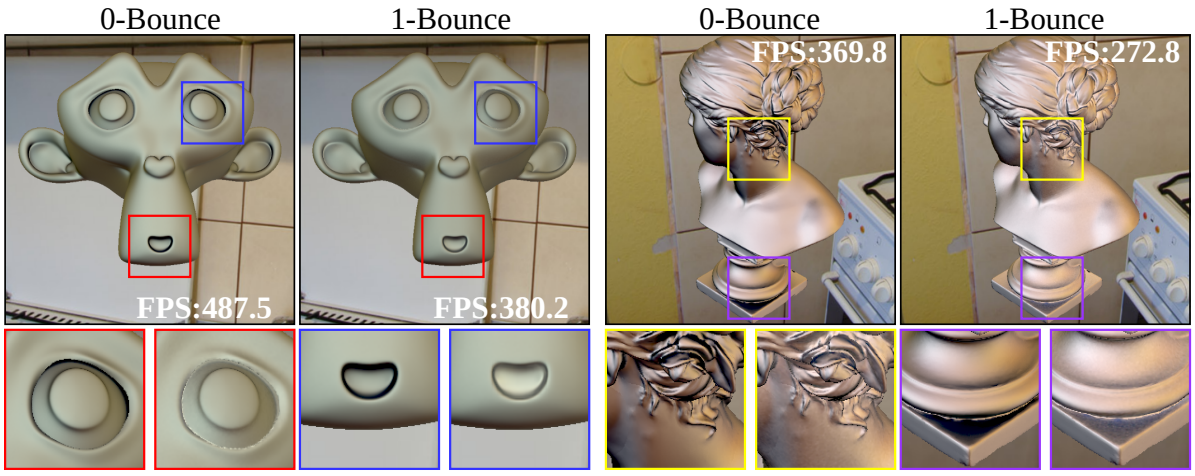


Figure 3.5: We demonstrate inter-reflections with transfer textures on two scenes: Diffuse Monkey (left) and glossy Roza (right). Renders with inter-reflection maintain real-time frame-rates, albeit slightly lesser than zero-bounce renderings.

Next, we demonstrate inter-reflections using transfer textures with the method described in Sec. 3.1.2. The zero-bounce and one-bounce renders with their corresponding FPS are shown in Fig. Fig. 3.5 for two scenes: *Monkey* and *Roza*. Because of extra texture fetch, convolution and evaluation operations the FPS with inter-reflections is slightly lower, albeit still real-time. As described in Sec. 3.1.2, additional bounces can be added with additional pre-computed textures.

Figure 3.6: **Normal Maps:** Transfer textures make it possible to use the shading normals from normal maps instead of the geometric normals. This is difficult to achieve with traditinal vertex based PRT. The above scene shows a minimally tessellated ground plane (wireframe, right) with a normal map. The scene is rendered with TPFL and transfer textures. All normal map details are preserved.

3.3.2 Normal Maps

Transfer textures make it possible to use normals maps during precomputation. This translates to lesser vertices during rendering as finer detail can instead be embedded in the normal map. Consider precomputation in traditional vertex based PRT. In this case if a normal map is applied, it only ever affects the transfer at those vertices thus loosing detail within each face when using high frequency normal-texture. With transfer textures we can output the *shading normal* from the normal map instead of the *geometric normal* in the G-buffer during precomputation. In Alg. 1 line 6, the transfer will then be computed at the shading normal instead. Since this texture is used to fetch transfer during rendering, all normal map details are preserved. We show renderings with normal maps in Fig. 3.6. The detail on the floor is due to the normal map without any additional vertices, as can be seen in the wireframe insets.

3.3.3 Memory Requirements

McKenzie et al. demonstrated textures with diffuse PRT using the formulation of Sloan et al. . Directly implementing glossy formulation by Sloan et al. with textures amounts to storing a $k \times k$ matrix per texel which quickly becomes intractable, even for reasonably small textures. Thus augmenting the triple product formulation to transfer textures is a clear choice. The memory requirements for vertex as well as texture (fragment) based approaches is shown in Tab. 3.2. The former's memory requirements depend on the scene complexity whereas it is constant for textures. Furthermore, a direct extension of Sloan et al. 's method to textures is infeasible, as shown in the fifth column (2.5 GB per texture).

Table 3.2: Memory requirements for vertex based and transfer texture based approaches for a 1024×1024 texture. Note that McKenzie et al. uses Sloan et al. 's approach which results in large textures for glossy rendering.

Scene	**# tris.**	**Vert. Mem.**		**Tex. Mem.**	
		Sloan et al.	Ng et al.	McKenzie et al.	Ours
Room	21K	64MB	2.5MB	2.5GB	100MB
Dragon	1.3M	5.2GB	215.8MB	2.5GB	100MB
TRM	441K	2.3GB	139.3MB	2.5GB	100MB
Plants	18K	64MB	2.5MB	2.5GB	100MB

3.3.4 Lower Bound on FPS

Since transfer textures are used in fragment shaders with an early depth pass, we achieve a lower bound on the FPS. The computation is roughly the same for each fragment and the worst case is when all fragments contain some geometry to be processed and rendered. This is in contrast to vertex based approaches, where run-time depends on the number of vertices in the scene. We demonstrate this in Fig. 3.4 in the *TRM* (441K verts) and *Dragon* (1.3M verts) scenes. Here, the FPS is lower for vertex based approach as compared to fragment based approach with transfer texture, in both TP and TPFL.

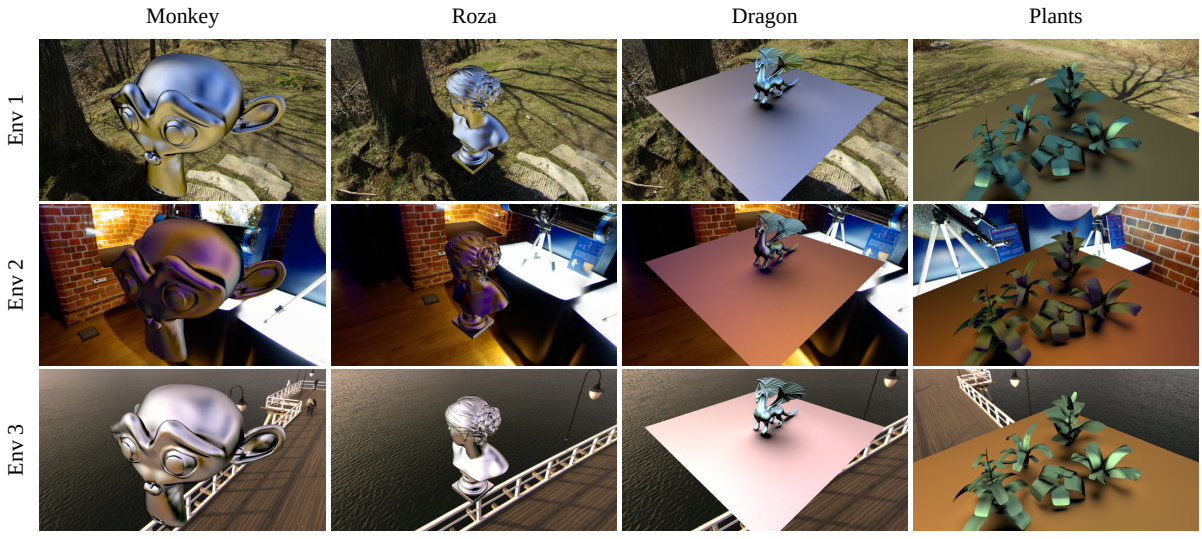


Figure 3.7: Results of our transfer textures method with different light settings

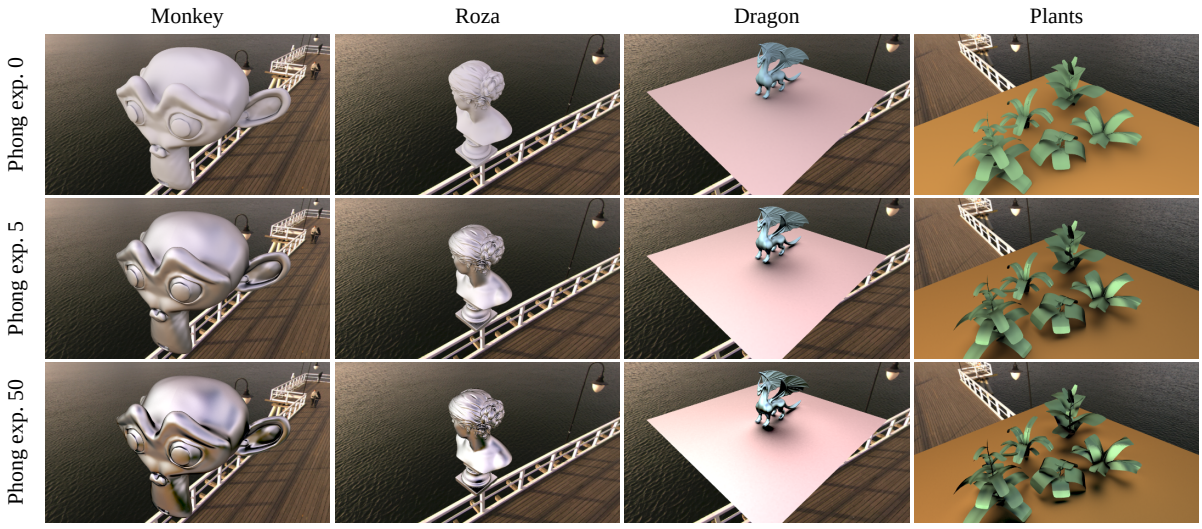


Figure 3.8: Results of our transfer textures method with different Phong BRDFs with same light

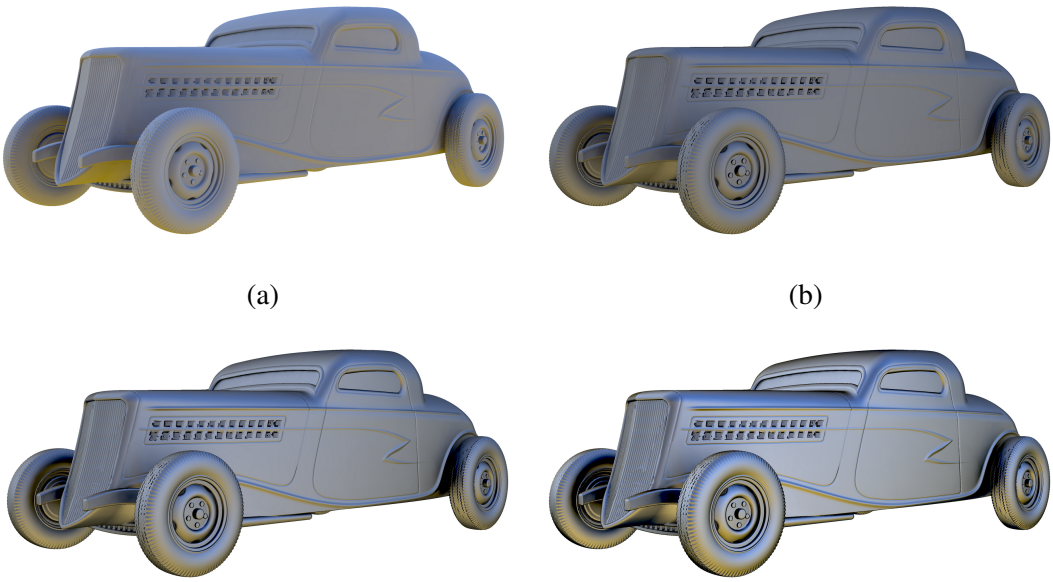
3.4 Conclusions, Limitations & Future work

In this paper, we presented *precomputed radiance transfer textures* for decoupling mesh tessellation from transfer sampling and storage for glossy rendering. We described methods to efficiently and correctly compute these textures and also demonstrated incorporation of inter-reflections using additional precomputed textures. We compared our renderings with traditional vertex based PRT approaches and thoroughly analyzed the memory requirements of transfer textures. We demonstrated real-time framerates for rendering with transfer textures on the fragment shader and superior render quality for minimally tessellated meshes. Additionally, we gave a lower bound on the FPS which will be useful in performance analysis in production. Our approach inherits the advantages of texture based optimizations like textures-sets, mip-maps and level of detail which can be easily incorporated. Although we demonstrate on a fixed texture resolution, it can be tailored accordingly depending on the hardware constraints and rendering quality needed. This is in contrast to vertex based methods that provide vertex count as the only control knob and little control over level of detail.

A limitation of transfer textures is that inter-reflections essentially bake the lighting and BRDF i.e. they cannot be changed without re-computation. We note that the work of [53] also bakes BRDF (including albedo) into their transfer matrices for inter-reflections. We would like to address this issue for future extensions of this work.

Chapter 4

Real-time Rendering of Arbitrary geometries using Learnt Transfer



(a) (b)

(c) (d)

Figure 4.1: The figure shows OLD-CAR defined as a Signed Distance Function (SDF) rendered with our approach for two different materials types: (a) Diffuse, (b)-(d) Glossy with increasing Phong exponent.

The use of textures to store sampled *transfer* function in PRT [53] provides artifact-free renders while leveraging the fast hardware accelerated texture lookups and barycentric interpolations. But, when using other geometric representations specifically, the ones which lack inherent storage schemas namely, *Vertex-attributes* and *Texture-mappings*. The extension of PRT is a very tedious process either requiring memory-intensive volume storages as in [14] or resampling-based FEM-based techniques like [27]. In contrast to the aforementioned approaches, we propose a learnt representation of transfer that can be

used with geometries that lack inherent storage schemas. We do this by learning a regression function that regresses *transfer function* from the input surface properties. For instance, the use of SDFs (Signed distanced fields) surfaces for rendering with traditional PRT methods is not possible as SDFs do not contain vertices or texture mappings to store and fetch transfer functions. Hence we propose the use of MLPs to regress the transfer values. Our method plausibly renders soft shadows and glossy highlights and achieves real-time framerates. We provide GLSL and CUDA based pipelines to obtain real-time renders.

4.1 Method

In this section, we first motivate and describe our learnt representation of transfer. Next, we discuss training details along with training data generation which ensures that our learnt representation is continuous. Finally, we describe the GLSL and CUDA implementations of our learnt representation. We evaluate and show the results of both these implementations in Sec. 4.2.

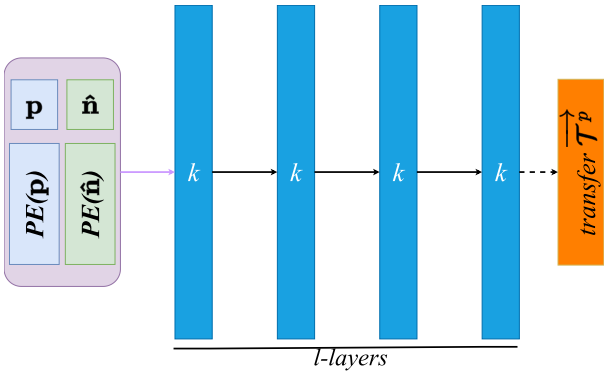


Figure 4.2: Our network is a Multi-Layer Perceptron (MLP) with k neurons per layer and l layers. A scene point p and its corresponding normal ($\hat{n}$) are fed to the network as input with positional encoding. The network outputs a vector, which are SH coefficients of *transfer*. The black-arrow lines represent a *leaky-relu* activation while the dotted arrow represents a *tanh* activation. Scenes rendered in this paper use $k = 64, l = 4$ or $k = 128, l = 4$ unless otherwise specified.

4.1.1 A learnt representation of transfer

The PRT formulation in Eq. (2.2) describes the evaluation of direct lighting at every point p. Ideally, it needs to be pre-computed at every visible point in the rendered view requiring transfer-vectors ($\overrightarrow{T^p}$) at all surface points of geometry. This is practically impossible. Instead, we learn a continuous function on the surface of the scene geometry as a function σ

$$\sigma : (p, \hat{n}) \rightarrow \overrightarrow{T^p} \tag{4.1}$$

that maps surface position(p) and normal($\hat{n}$) to a transfer value. We use a Multi-Layered Perceptron (MLP) network as σ. Neural networks are known to be universal function approximators [19]. Recently, neural representations using coordinate-base MLPs have found a great utility to learn the distribution of radiance in a scene [13, 35, 48, 67]. We adapt this idea and learn σ as an MLP to represent a continuous function over the given scene surface. We train the MLP by precomputing the transfer values at densely sampled surface points. This method works on all types of surface representations: meshes, implicit surfaces, SDFs, parametric surfaces, etc.

Formally, given a scene point p and its normal $\hat{n}$, we train a MLP to learn a mapping function σ presented in Eq. (4.1) to learn the SH-vector of transfer $\overrightarrow{\mathcal{T}^p}$ with components $\mathcal{T}_i^p$ (Eq. (2.3), Eq. (2.2)). In our experiments, we use 16 SH coefficients similar to [64], making $\overrightarrow{\mathcal{T}^p}$ 16 dimensional. The network architecture used is shown in Fig. 4.2. It contains l layers with k neurons per layer. Each layer is followed by the leaky-relu [31] activation function, except the last which is followed by *tanh* activation. In practice, the network inputs are projected to a high dimensional representation using positional encoding [35, 44, 58]. We set $k = 64$ and $l = 4$ or $k = 128$ and $l = 4$ for scenes in this paper unless otherwise specified. Previously, [53] and [25] stored transfer values on mesh vertices. Textures in a *UV-space* were also used to store transfer values [10, 34]. These strategies strongly rely on dictionary storage structures inherently present in mesh-based geometric representations and do not extend to implicitly represented scene objects. They also incur high memory costs to store transfers for complex scenes.

4.1.2 Training details

Our training dataset consists of transfer values for densely sampled points in the scene. We ensure an equal distribution of these points by an area-based sampling of the mesh surface [60]. The ground truth transfer is calculated using ray-tracing for each sampled point, as is done traditionally [46]. Note that training data can be generated for any other geometric representation by first converting it to a mesh and then applying the above routine. For example, in the SDF case, we use marching cubes to first extract a mesh, densely sample the mesh using the above method and project these points to the SDF surface. The Transfer is then calculated for these points by sphere tracing the SDF. We project the resulting transfer to SH basis with 16 coefficients which are used as ground truth. By training our network with dense and equally distributed samples, we ensure that it learns a continuous representation of transfer. For scenes used in this paper, we generate $800k - 1M$ points on the geometric surface and calculate the associated transfer vectors to train our network. The data generation takes around 2 hours per scene. We use a batch size of 8192 and train our network with this data for around 200 epochs with the ℓ_1 loss. The time taken for training is about 10 minutes on NVIDIA RTX 3090 GPU. We use sampled points as a training set and try to regress on the vertex locations which are not included in the training sample to ensure the fitted function is consistent with the surface geometry. Our sampling scheme [60] ensures that the sampled points do not co-inside with the vertex locations whereby keeping the test set different from the train set ensures uniform fitting rather than an overfit of the region. Refer to Sec. 4.1.3. for more details about training and sampling.

4.1.3 Sampling

For sampling, we make use of the work by Turk et al. [60] method of uniform sampling over a surface of a triangle. This sampling technique ensures there is no skewness towards the vertices of the triangle. Usually, the sampling function used is

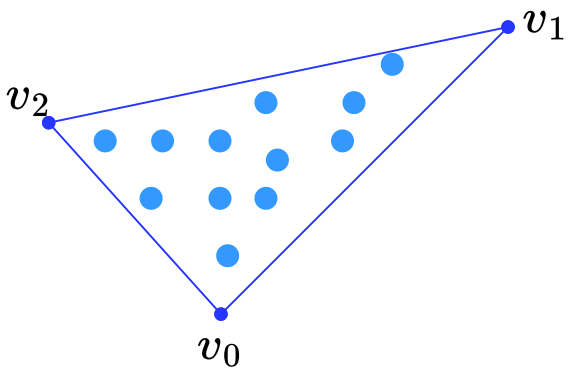


Figure 4.3: The image shows the samples of [60]. It can be seen clearly that the points are not skewed towards the triangle's vertices(v_0, v_1, v_2). They are uniformly distributed on the surface

$$(1 - \sqrt{\xi_1})v_0 + \sqrt{\xi_1}((1 - \xi_0')v_1 + \xi_0' v_2) \ni \xi_0', \xi_1 \in [0, 1) \tag{4.2}$$

ξ_0', ξ_1 are sampled using a random number generator. To further ensure samples not falling on vertices (v_0, v_1, v_2) we can constrain values ξ_0', ξ_1 to tighter boundaries in between $(0, 1)$. This ensures vertices and their vicinities are not included in the train set.

For sampling on a surface, we do a face weight sampling taking area as the metric. We had to implement our own sampler rather than using one in Trimesh [9] so as to leverage the sampling strategy of [60] (Refer to Fig. 4.3).

4.1.4 Real-time rendering with GLSL & CUDA

Once trained on a given scene, our network can output transfer vectors for *any* point in that scene. Furthermore, since our network is small, it allows for efficient per-pixel evaluation on the GPU. Below we discuss two different implementations, one with GLSL and the other with CUDA.

4.1.4.1 GLSL

We implement the GLSL version within the ModernGL framework [11]. The network weights and biases are hardcoded in 4×4 matrices (**mat4** type) in the fragment shader. Specifically, the weights of each layer are divided to fit into multiple **mat4**s. This allows for an efficient forward pass using matrix-vector products on the GPU. We have automated this using a script-based extraction of weights into

the shader. With the network weights available as matrices within the shader, the rendering proceeds as follows. We first obtain the surface position p, its normal $\hat{n}$, and the view vector ω_o in the fragment shader using the standard OpenGL pipeline. An early depth pass is used to avoid the processing of unnecessary fragments. Next, we apply positional encoding [44] to p and $\hat{n}$ and evaluate the forward pass of our network with the hardcoded weights. This results in the SH vector of transfer $\overrightarrow{\mathcal{T}^p}$ at p. The transfer is used with the triple product formulation [39] with a global light matrix to obtain the shading at p according to Eq. (2.3), Eq. (2.2). Our GLSL implementation works for both AMD and NVIDIA GPUs since OpenGL itself is supported on both hardware architectures. A downside of this implementation is that the network size is constrained by the amount of local memory available for each shader.

4.1.4.2 CUDA

To alleviate the dependence of network size on local shader memory, we implement the second version in CUDA. Before rendering starts, we pre-load the trained weights of our network on the GPU. Rendering is then performed in two separate passes: (1) An OpenGL pass to extract G-buffers, (2) A dedicated CUDA pass for network forward evaluation using the pre-loaded weights. The OpenGL pass works similarly to the GLSL implementation, except it only extracts G-buffers and does not perform shading. We create CUDA buffers that point to the resulting G-buffers from the previous step, which implies that these share the same GPU memory. These CUDA buffers are then converted to PyTorch [42] tensors using the PyCuda [24] interface. This approach avoids expensive transfers between GPU and the host and is crucial to achieving real-time framerates. The PyTorch tensors are then used for the network forward pass with positional encoding. The network evaluation only processes parts of the G-buffers that intersect geometry by packing them before the network forward evaluation. The transfer vectors obtained from the network are then unpacked to their original locations in the G-buffer. This avoids unnecessary evaluations of the network further improving the FPS. The final image is obtained by combining this transfer with global light matrices with the triple product formulation [39], similar to the GLSL pass. This implementation allows the network to be as large as the entire GPU memory since it has a dedicated forward pass for it. Refer to Fig. 4.4 for a visual explanation of this implementation and a caption for details.

4.1.5 Handling Large Scene without loss of performance

In certain scenarios, while handling large scenes, a small neural network might not be sufficient to regress the *transfer*-accounting for the cosine weighted visibility of the rendering equation. In such a scenario, we can either engage a large neural network or subdivide the scene into smaller segments and assign a small network to regress the respective transfer values. The latter is a *more performance-friendly* approach as the number of operations required to calculate transfer will be substantially smaller than the former. This is due to the fact that per fragment only a small MLP needs to be evaluated as opposed to a large network, resulting in real-time performance obtaining render-framerates of over

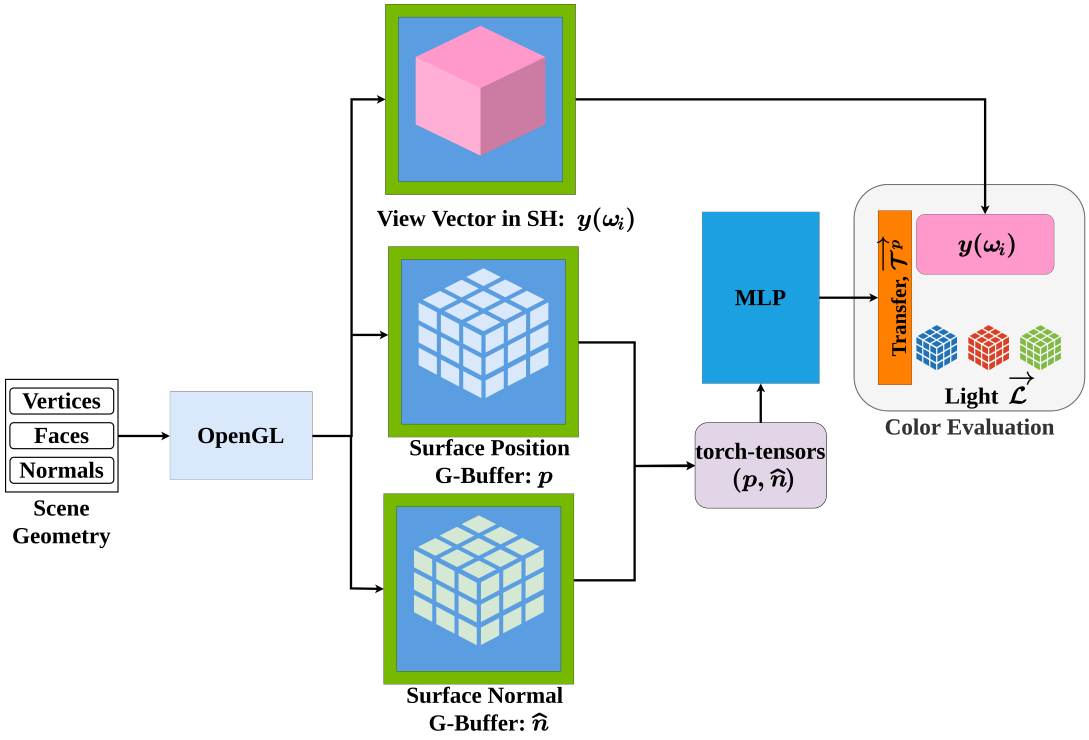


Figure 4.4: CUDA Implementation: We extract G-buffers that store the position p and normal $\hat{n}$ in texture space. Along with G-buffers we also project per fragment view directions to SH and store them in textures. All OpenGL Textures (■), are shared with CUDA buffers(■). The CUDA buffers are mapped to the same memory location as OpenGL texture to avoid expensive GPU - Host transfers. The CUDA buffers are copied to preloaded torch-sensors(■) residing on GPU. The tensors are then fed to the network residing on GPU and the output is obtained. Since the operations are only between GPU-GPU without involving the host we avoid host latency. The network outputs a transfer vector which is clubbed with lighting and SH representation of view direction extracted as torch tensor to obtain color at each pixel using the Triple Product Formulation [39]. Note: Before passing the G-buffers to MLP, fragments that intersect the geometry are separated from the ones that do not and packed to avoid unnecessary network computations.

200FPS. To achieve this we subdivided the scene into sub-scenes and sample points in each sub-scene using the strategy explained in Sec. 4.1.2. To maintain continuous seamless regression of transfer at boundaries we sample points for extra δ area over the adjacent sub-scene refer to Fig. 4.5 for details. For each sample point, we trace rays into the un-subdivided(whole scene) scene. This ensures the visibility accounting for the full scene rather than the sub-scene. Once transfer vectors are calculated, to ensure a minimal number of MLPs being used to regress the transfer of the whole scene we cluster neighboring sub-scenes using Variances analysis of the transfer vectors similar to the techniques of CPCA [54]. Contrary to their method, we do not use Principle components rather just rely on the small MLPs to regress transfer. At run-time, every sub-scene is assigned respective MLP either in GLSL or CUDA based implementation. Thus, enabling parallel execution of the fragments achieving real-time framerates of 200FPS. A visual representation of the above strategy is given in Fig. 4.6.

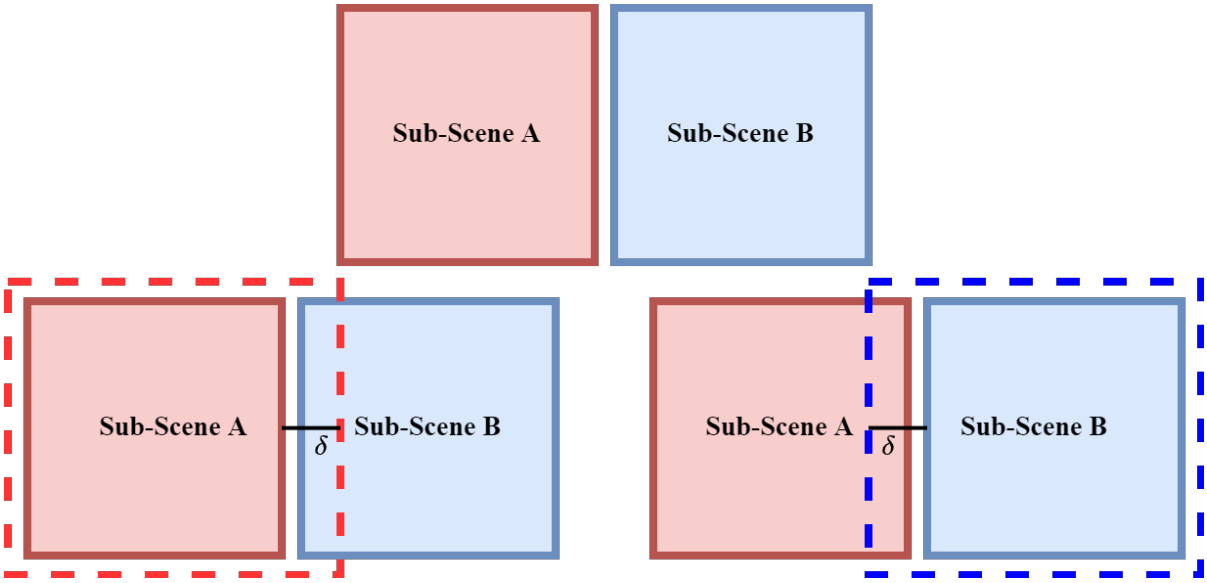


Figure 4.5: Sampling at boundaries: While handling the boundary regions of neighboring sub-scenes which are caused due to the sub-division. We sample δ area more into the adjacent sub-scene and include the sample points falling into that δ region into the training set of the sub-scenes MLP. This ensures smooth learning of transfer vectors and avoids seam artifacts.

4.2 Validation & Results

In this section, we validate and show the results of our learnt representation (Sec. 4.1.1) implemented in GLSL and CUDA (Sect. Sec. 4.1.4). Our network is trained for each scene and we generate training data as outlined in Sec. 4.1.2. We compare our renderings with baseline PRT, which uses texture-based storage of transfer [10, 34] with the triple product formulation [39]. We use the texture version as the

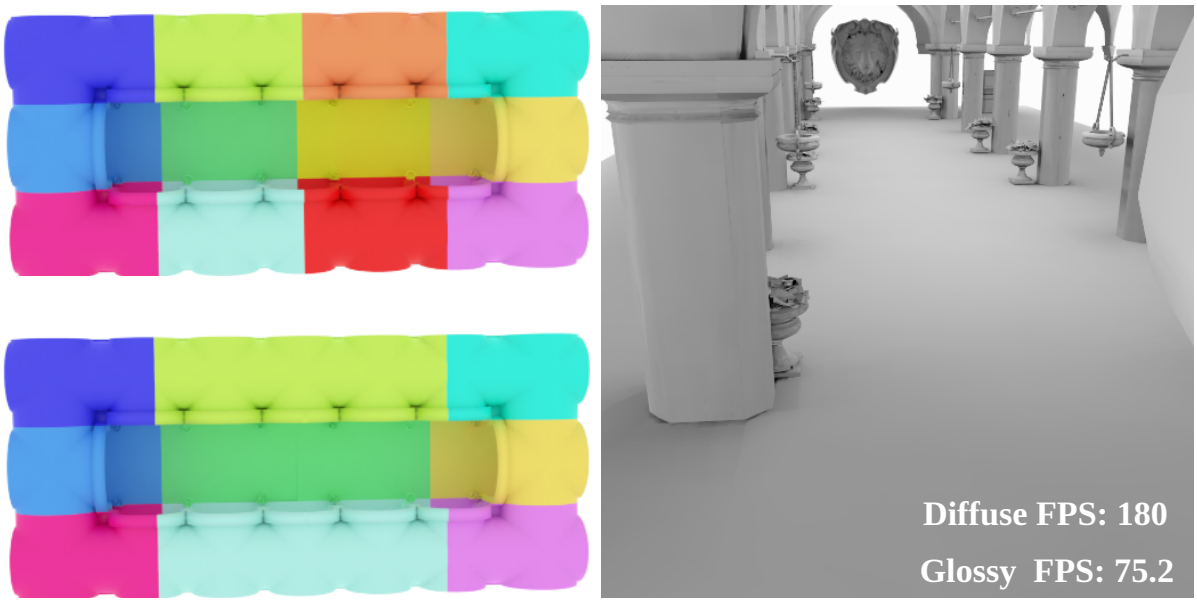


Figure 4.6: Large Scene: The Figure on the *top-left* shows the top-view of SPONZA Cathedral divided into 12 even parts. We use the heuristic presented in Sec. 4.1.5 to join the few sub-scene to reduce the number of networks required to regress the Transfer. The clubbed geometries are visualized in the *bottom-left* image where the components of the middle section are joined with their respective neighbors. Please observe that both *top-left* and *bottom-left* images are color-coded, each unique colors represent a sub-scene. We utilize the *bottom-left* configuration of the sub-scene which only requires 9 small MLPs rather than 12 as in the case of *top-left*. We visualized the resulting render in the *right*. It is to be noted that we have used White light to show the transfer regressed from the network is independent of the Lighting and material of the object and can be swapped with ease as depicted in Fig.(Fig. 4.9,Fig. 4.1). FPS of respective materials are mentioned in Figure.

baseline since it produces artifact-free renderings in most cases and avoids the memory overhead of vertex-based approaches for dense meshes. The baseline implementation stores transfer in a 1024×1024 texture with optional texture-sets for large meshes. In Sec. 4.2.1, we validate our renderings of triangle meshes by qualitative and quantitative comparison with baseline PRT. In Sec. 4.2.2, we show results on scenes with SDF as the geometric representation. This is possible since our network learns a continuous representation of transfer and can predict it for any point in the scene. We show results on both analytical and neural SDFs. Lastly in Sec. 4.2.3, we compare and contrast our GLSL and CUDA implementations and discuss their framerates for different network sizes. All scenes are rendered using our GLSL implementation (unless otherwise specified) on an NVIDIA RTX 3090 GPU and at a resolution of 1024×1024.

4.2.1 Validation on triangle meshes

We validate our results on four scenes with triangle meshes: TEASET , PLANTS , ROZA , & DINING-TABLE . Additionally, we also show results on a large scene leveraging the strategy presented in Sec. 4.1.5 and Fig. 4.6 shows rendered results. These scenes contain high-frequency changes in visibility, for example in the hair of ROZA and shadows due to leaves in PLANTS . And our network is able to regress the SH vectors of transfer with ease. We render our results with diffuse and glossy materials using two different sizes of the network: $k = 64, l = 4$ and $k = 128, l = 4$ and compare them with baseline PRT. Rendering results are shown in Fig. 4.8 and quantitative metrics (MAE, PSNR, SSIM) along with frame times (FPS) are shown in Tab. 4.1. The quantitative metrics are calculated by comparing our rendering with baseline PRT as a reference. Our rendering results closely match the baseline and achieve high PSNR and SSIM values, which validates our method. We further compare our rendering of the diffuse ROZA scene with baseline PRT in Fig. 4.7. Our method is able to reproduce fine soft shadows which further strengthens our validation. As shown in Tab. 4.1, the $k = 64, l = 4$ network comfortably achieves real-time FPS for both diffuse and glossy materials. A larger network ($k = 128, l = 4$) achieves better rendering quality and metrics albeit at a drop in FPS.

4.2.2 Results on SDF

We now present our rendering results on scenes with SDF geometry. As stated earlier, implicit representations like SDFs do not have an inherent storage schema like the Mesh, which makes it very hard to store transfer vectors, hence we utilize the MLPs we trained to regress transfer. Rendering is done using the GLSL implementation (Sec. 4.1.4.1). We sphere trace the SDF and obtain transfer at the intersection point using our network. Since sphere tracing needs to be done for all fragments, the scene geometry is set to two triangles that cover the entire image. We show results on four analytic SDFs, MIKE-MONSTER , RABBIT from [12, 43] and OLD-CAR , FISH from [56]. We also show results with one Neural SDF, STANFORD-BUNNY using the method of [8].

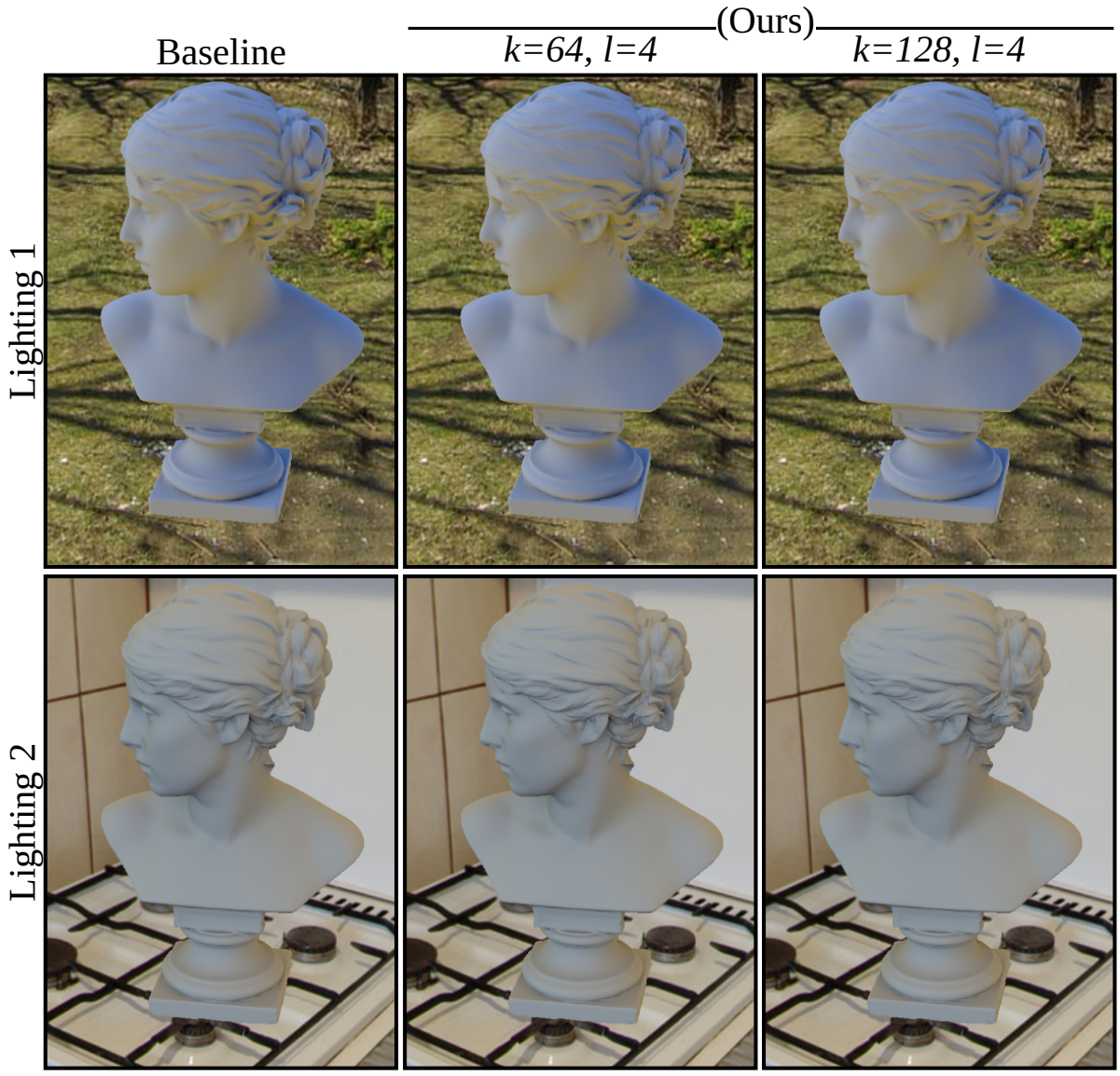


Figure 4.7: Diffuse Mesh Results: Since our learned function is only accounting for transfer we can dynamically change the light on the fly without retraining the network. We show results of ROZA with two different lightings while using the same set of learned weights. Our approach plausibly renders soft shadows, especially in high-frequency visibility changes in the hair.

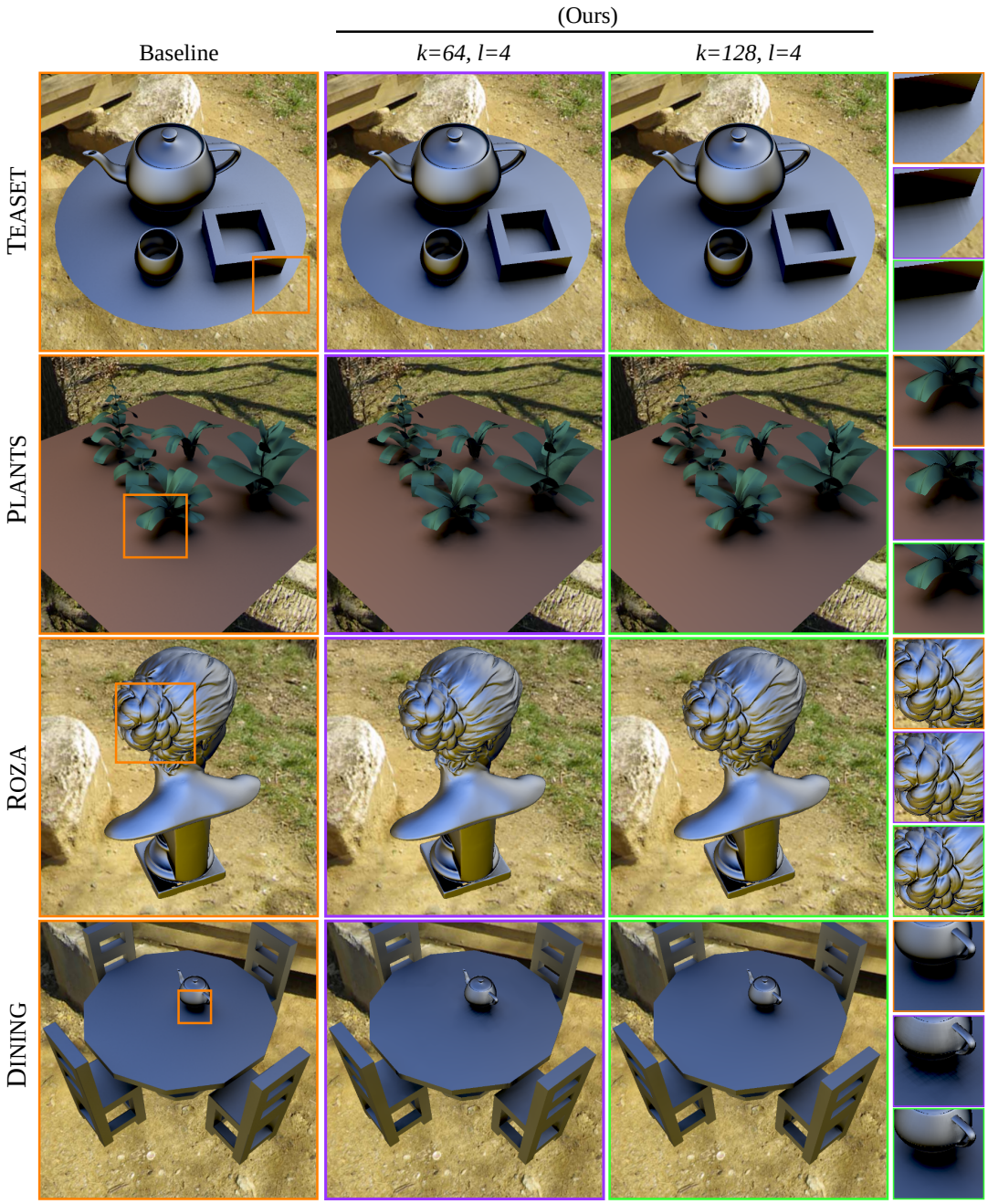


Figure 4.8: We show the results of our approach and compare them with the baseline (left), which is a texture-based transfer storage implementation of PRT. Our results are rendered with $k = 64, l = 4$ network (middle) and $k = 128, l = 4$ network (right). Insets are provided at the right-extreme, where we can observe that $k = 64, l = 4$ network produces plausible results albeit with a few jaded lines in the case of PLANTS and DINING-TABLE , while $k = 128, l = 4$ network gets rid of those artifacts and matches the baseline. We note that $k = 64, l = 4$ network comfortably achieves real-time FPS while $k = 128, l = 4$ has a lower FPS (Tab. 4.1). *Please inspect insets closely*

Diffuse								
Scene	*k=64, l=4*				*k=128, l=4*			
	MAE	PSNR	SSIM	FPS	MAE	PSNR	SSIM	FPS
TEASET	0.00074	49.227	0.99708	350.5	0.00055	51.050	0.99757	35.1
PLANTS	0.00109	44.450	0.99399	201.2	0.00076	46.290	0.99573	20.5
ROZA	0.00123	44.068	0.99462	330.2	0.00080	48.261	0.99658	38.1
DINING-TABLE	0.00168	41.917	0.99066	300.0	0.00106	43.494	0.99280	40.1
Glossy								
Scene	*k=64, l=4*				*k=128, l=4*			
	MAE	PSNR	SSIM	FPS	MAE	PSNR	SSIM	FPS
TEASET	0.00133	45.752	0.99356	105.5	0.00102	48.129	0.99504	25.1
PLANTS	0.00319	35.701	0.98237	64.5	0.00250	36.614	0.98661	12.2
ROZA	0.00122	42.449	0.99701	120.5	0.00083	46.768	0.99817	31.1
DINING-TABLE	0.00411	32.638	0.97393	130.1	0.00312	33.414	0.97886	32.1

Table 4.1: This table shows comparison among two different network sizes used to learn the transfer function. The quantitative metrics (MAE, PSNR & SSIM) are calculated by comparing our rendering with baseline PRT as reference, which is a texture storage implementation of PRT. These metrics averaged amongst 30 uniformly sampled views in a trajectory for both DIFFUSE and GLOSSY material configurations. We also show the FPS obtained for respective network sizes. Our approach renders in real-time for both diffuse and glossy configurations.

Scene	FPS			
	k=64, l=4		*k=128, l=4*	
	Diffuse	Glossy	Diffuse	Glossy
OLD-CAR	254.96	207.28	73.2	38.8
MIKE-MONSTER	294.28	173.6	61.0	24.2
FISH	320.1	231.6	77.1	50.1
RABBIT	310.21	205.94	73.2	37.9

Table 4.2: Performance on diffuse and glossy renders of analytic SDFs. Our approach can render SDFs within the PRT framework in real-time.

Transfer / Surface	*k=64, l=4*		*k=128, l=4*	
	Diffuse	Glossy	Diffuse	Glossy
k=32, l=2	320	200	77	20.1
k=64, l=2	50	25	26.5	15.2

Table 4.3: FPS on the STANFORD-BUNNY Neural SDF (Fig. 4.10).

Renderings with glossy materials using our method are shown in Fig. 4.9. We render these scenes with two different environment lighting. All highlights from the environment map are reproduced on the glossy surface accurately. For instance, the *yellowish* ground in the environment map reflects at the bottom of the SDF and the *bluish* sky reflects at the top. We also show renderings of one diffuse and three glossy results with increasing Phong exponents on the OLD-CAR SDF in Fig. 4.1. All scenes produce real-time FPS while producing plausible glossy highlights and soft shadows as shown in Tab. 4.2, Fig. 4.9, Tab. 4.2. The FPS for SDFs behaves similarly to triangle meshes, in that the $k = 64, l = 4$ network comfortably achieves real-time FPS while the $k = 128, l = 4$ network has a lower FPS.

Finally, we show our rendering of the STANFORD-BUNNY scene, which is defined as a Neural SDF as a proof of concept. We use the SDF optimized by [8]. The result is shown in Fig. 4.10 for diffuse and glossy material and two different lighting conditions. We also show the FPS in Tab. 4.3 for two different SDF networks sizes (vertical) and two different sizes of our network (horizontal). The renderings are plausible and achieve real-time framerates for smaller network sizes while maintaining interactivity for larger network sizes. The FPS could be further improved by optimizing the tracing of SDFs [57], a research direction orthogonal to ours.

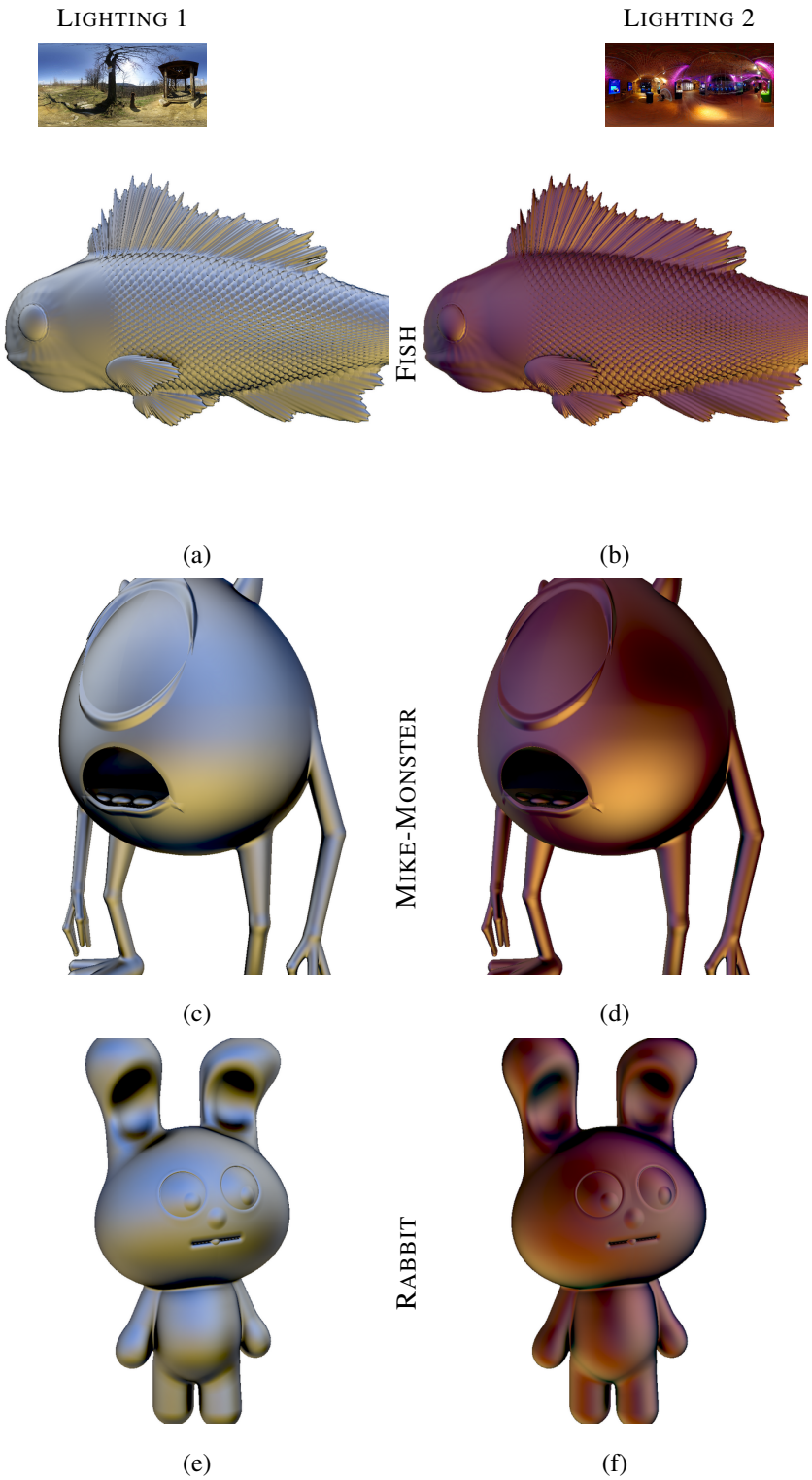


Figure 4.9: SDF Results: With the use of learnt transfer approach we extended PRT onto SDF where UV mapping or vertex storage is not defined. The learnt function approximates the transfer well. Results are shown on three SDFs: MIKE-MONSTER , FISH , RABBIT with two different lighting environments.

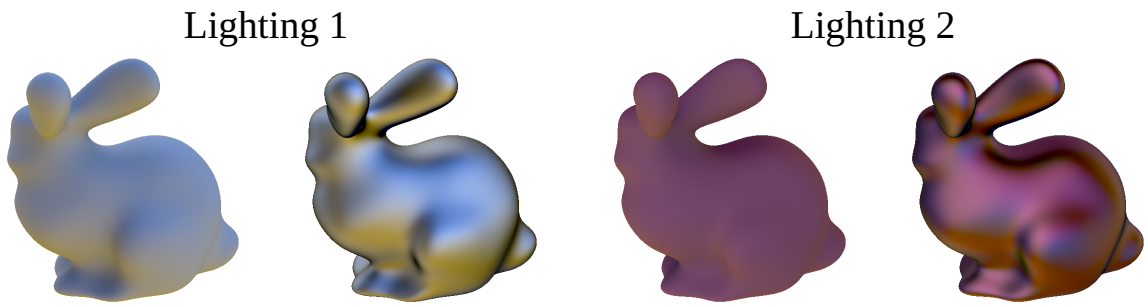


Figure 4.10: We show renderings of the STANFORD-BUNNY neural SDF learnt using two network architectures. We show diffuse and glossy renderings of the neural SDF by evaluating our learnt transfer on the surface. Note that the geometric representation of STANFORD-BUNNY is obtained from the [8] and we only contribute learnt transfer to it.

4.2.3 GLSL & CUDA: Comparison

We compare and contrast the GLSL and CUDA implementations of our learnt representation. As mentioned before, the GLSL implementation limits the network size to the available local shader memory. On the other hand, CUDA allows the network to be as large as the entire GPU memory. We experimented with four different network sizes for both implementations: $k = 64$, $k = 128$, $k = 256$ & $k = 512$ with $l = 4$. The FPS for these configurations on the PLANTS scene are listed in Tab. 4.4. The GLSL implementation outperforms for $k = 64$ network, while the performance drops on higher sizes. It can be observed that CUDA starts to take the lead from $k = 128$ network. In fact, for $k = 256$ and above, GLSL implementation fails to render. Although the performance of the CUDA implementation drops for higher network sizes, it is still able to load and render the scene interactively.

Implementation	$k = 64$	$k = 128$	$k = 256$	$k = 512$
GLSL	64.4	12.2	N/A	N/A
CUDA	55.1	29.1	20.7	8.0

Table 4.4: CUDA vs GLSL: Table contains the Performance in FPS of the GLSL and CUDA implementation with various MLP architectures ranging from $k \in (64, 512), l = 4$ tested on PLANTS scene with a glossy material. The GLSL implementation outperforms the CUDA implementation for $k = 64$, but degrades at larger network sizes or fails to render ($k = 256, k = 512$). The CUDA implementation renders in all cases and is real-time for $k = 64$ while maintaining interactivity for other cases. Please note we use the same number of layers($l = 4$) in all the architectures presented in the table

4.2.4 Memory requirements

Thanks to our simple network structure, the entire transfer of a scene can be represented as a set of weights. In vertex-based storage of transfer as in Sloan et al. , the memory requirement has a one-to-one correspondence with the mesh tessellation. With texture-based storage, this correspondence is reduced, with the caveat of needing a good UV-mapping or possibly a large texture. With our method, this correspondence is further reduced as the entire transfer is now just a set of weights. For example, a mesh with $1M$ vertices requires $\approx$ 70MB data stored on vertices, while our method only needs around 64KB - 5MB. On the other-hand grid-based storage requires 256MB of data for a grid resolution of 512^3.

4.2.5 Visualization of regressed transfer

As discussed earlier our representation learns a faithful mapping of surface to *transfer* values. Since transfer ($\mathcal{T}_i^p$) is a cosine weighted visibility, which is a spherical function, we can project it back to the spherical canvas. We visualize the ground truth transfer calculated using ray-tracing Vs the one regressed by our MLP in Fig. 4.11

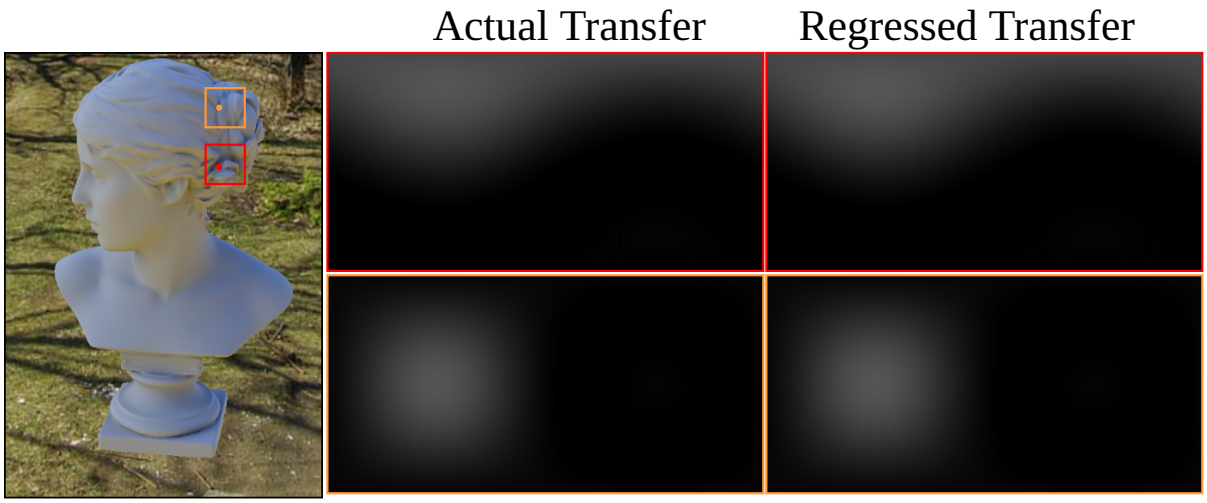


Figure 4.11: In this figure we visualize the *transfer* by back projecting it into the spherical space for both ground truth calculated via TexturedPRT and one which is regressed using the small MLP. It can be seen clearly that for both points as visualized in (left) Roza the regressed transfer is faithfully matching with the ground truth *transfer* of TexturedPRT.

4.3 Additional Results and Ablations

4.3.1 Additional Mesh Results

In this section, we show more results specifically on another large scene. For this, we chose a LIVING-ROOM with a sofa, floor, and three shelves. The scene can be divided based on bounding boxes as shown in Fig. 4.14(a) or semantically as shown in Fig. 4.14(b). Either way, we can assign each color-coded sub-scene a different MLP to learn and regress transfer. As we have chosen to use the bounding box method earlier, we want to demonstrate the semantic sub-division of the scene here. Each sub-part *Sofa*, *right-shelves*, *flooring* and *front and top shelves* are sampled using the method presented in Sec. 4.3.1. The so-obtained renders are demonstrated in Fig. 4.15. We show results of the same view with two different lighting conditions while keeping the albedo constant. Respective FPS and Lighting conditions are represented as insets in Fig. 4.15. Please feel free to zoom in and verify the shadowing and lighting details.

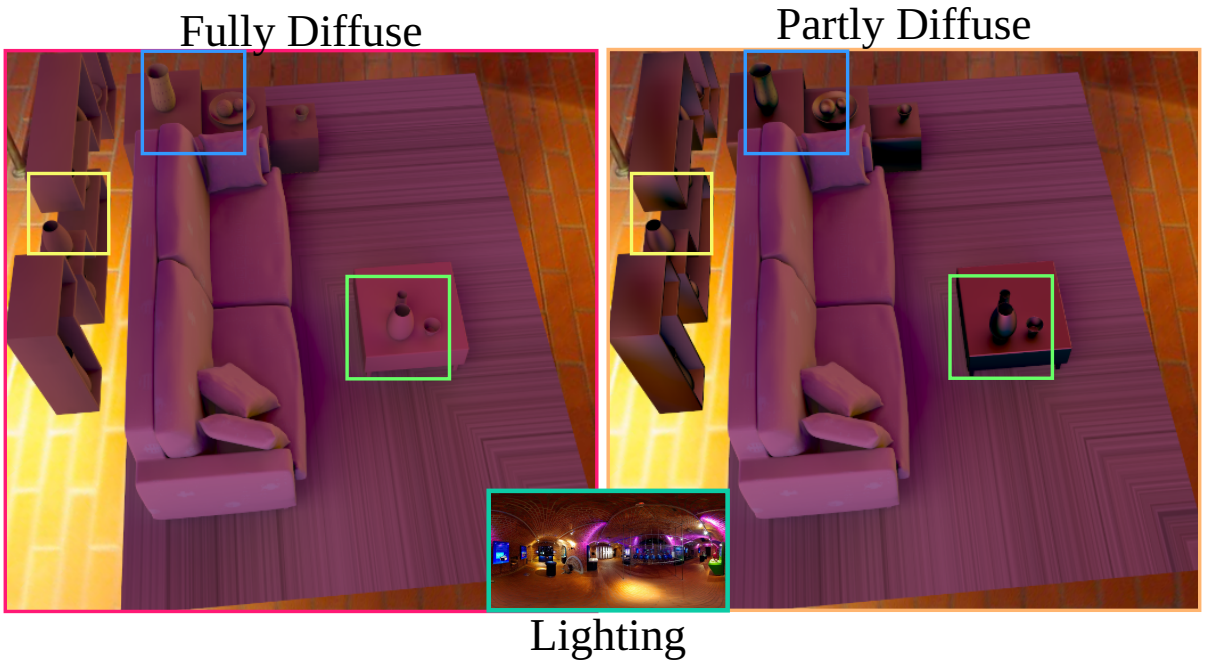


Figure 4.12: The left shows a fully diffuse scene, while the right shows partly diffuse with all the *shelves: front, side, and top* being glossy. Both are lit using purple lighting as shown in the inset. Please observe the inset which demonstrates shadow effects being faithfully reproduced.

Apart from changing the material of the entire scene, we can also partly change the material of the object. In Fig. 4.12 left we show a fully diffuse scene. On the right, all *shelves: front, side, and top* are made glossy while keeping the floor and sofa still diffuse.

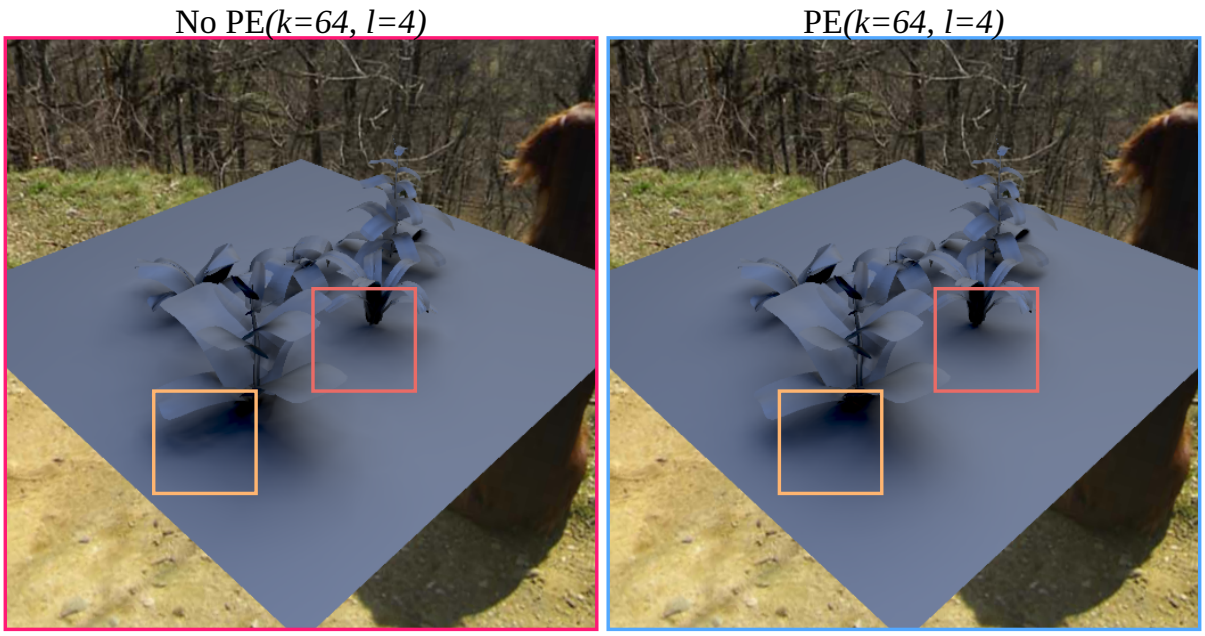


Figure 4.13: In the left we show results of MLP without positional encoded inputs. On the right, we show the results of positional encoded inputs. Observe the highlighted regions where the visibility changes are not properly captured in the case of inputs with no positional encoding. While the ones with positional encoding produced the results faithfully. Please note that we explicitly disabled albedo so that shadows can be clearly observed

(a) (b)

Figure 4.14: Sub division: As we have geometry at hand we can sub-divide the scene semantically or using a bounding box. (a) shows a bounding box-based division, while (b) shows semantic division.

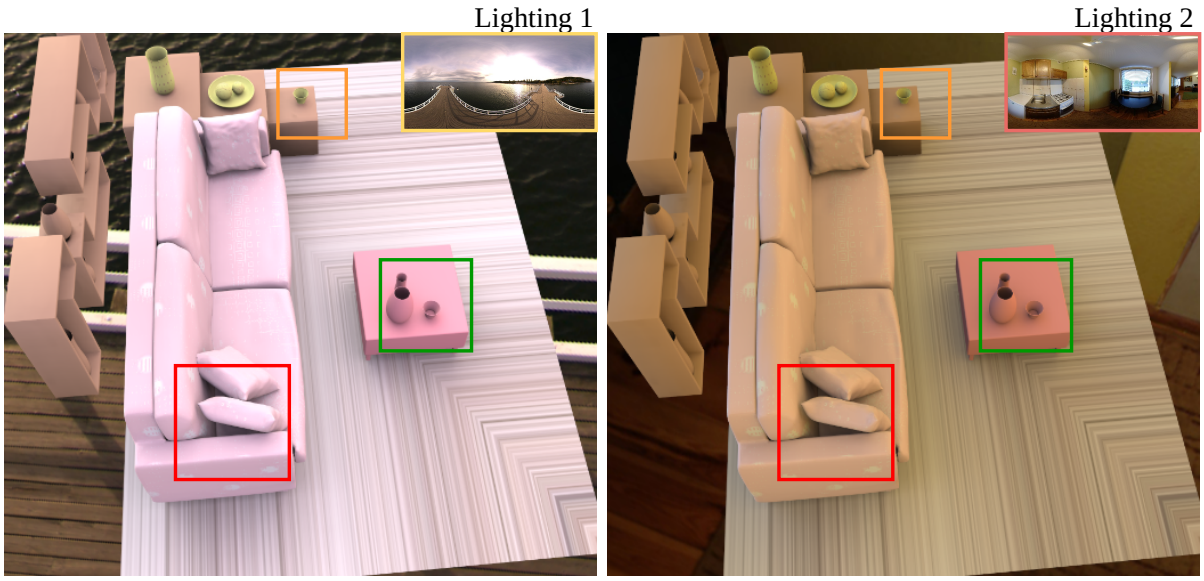


Figure 4.15: The figure shows the large scene LIVING-ROOM scene with two different lighting conditions. Please observe the insets, which accurately produce ray-tracing effects of direct illumination. The FPS numbers are mentioned below the figure. Please note that lighting change does not cause FPS drops. The drop is due to the glossy renders requiring a Triple Production evaluation as discussed in Sec. 2.2

4.3.2 Choice of Loss function And Effect of Positional Encoding

Loss Function We have experimented with different losses including L_2 and projection loss.

L_2 Loss: The L_2 loss was not able to capture the minute differences in the SH-vectors leading to missing shadows.

Projection Loss: In the case of projection loss, we projected the SH-vectors into Spherical Canvas and tried taking L_2 loss on ground-truth vs regressed transfer. This was smoothing out the transfer vectors due to the fact that projection was limited to the resolution of the canvas. The higher resolution of Canvas leads to an OOM(Out Of Memory). Hence, chose to go with L_1 loss.

Architecture and Positional Encoding: Since we aim to obtain real-time FPS while producing plausible renders we need to limit the number of neurons per layer (k), ensuring forward the evaluation is faster. Hence we have experimented with $k = 64, 128, 256$ and found that $k > 128$ fails to maintain real-time FPS. The Tab. 4.4 shows this behavior. When experimenting without Positional Encoding [35]. We have observed artifacts in cases where transfer varies sufficiently over the surfaces. The visual description is in Fig. 4.13.

4.4 Discussion & conclusion

In this paper, we presented a learnt representation of transfer through small MLPs in traditional PRT frameworks. Our main motivation was to alleviate the dependence of transfer storage in PRT on geometry representations. We ensured that our network learns a *continuous* representation of the transfer in a scene by densely and equally distributing training samples over the surface. We provided a strategy to handle large-scenes in Sec. 4.1.5 and Fig. 4.6 while maintaining real-time frame-rates. We demonstrated two implementations of our learnt representation: GLSL and CUDA. We analyzed both implementations and discussed their merits and drawbacks. We further demonstrated that both implementations achieve real-time FPS on meshes as well as SDFs. SDF rendering which we demonstrated with our approach was not easily possible within the PRT framework. Utilizing our approach the works [63, 64] can incorporate SDF geometries into their scenes.

One interesting challenge would be to empirically determine the optimal size of the network for a given complex scene, we would like to work on it in the future. We would also like to study the benefit of representing other surface properties, like specularity and albedo with our proposed approach. Finally, we would like to investigate the incorporation of inter-reflections with our approach, without baking in BRDF as done in [53], for real-time global illumination with PRT with *flexible surface representations*.

Chapter 5

Conclusions and Future Work

In this thesis, we have explored the technique of Precomputed Radiance Transfer (PRT) for various geometric representations while maintaining real-time framerates. Through the thesis, we first explained prior work on 1) how the rendering equation is disentangled under certain assumptions into various components like Lighting and Visibility, 2) how they are computed, and 3) clubbed together to obtain final radiance, followed by the design choices we make and additional functionalities we add to the existing works.

We first discussed the necessity of smooth sampling of the Transfer function for an artifact-free representation. Unlike prior art which stores Transfer at vertex locations, leading to unnecessary tessellation, the work of Chapter 3 presents an idea of storing Transfer at UV-mapped locations. Additionally, along with the storage of Transfer accounting for cosine weighted Visibility, the work also provides a way to store the Irradiance maps of the 1-bounce reflections efficiently in the UV space extending the Triple Product Formulation [39]. The method produces real-time rendering framerates of about 70FPS on commodity hardware at $1\text{K}\times1\text{K}$ resolution.

When handling different storage spaces for Transfer we have observed the Transfer function takes the assumption that an inherent storage schema is *always present in the underlying geometric representation* namely Mesh vertices, UV maps, etc. More specifically, the UV mapping needs to be area-preserving and meshing needs to be dense to achieve artifact-free renders. These schemas are not present in Implicit Surfaces like SDFs, hence to extend the PRT framework to arbitrary geometric representations we present an idea of learning small MLP which can regress spatially varying Transfer signals based on the surface properties like position and gradient (normals). To maintain the desirable feature of PRT which is "rendering at real-time framerates", we provide two methodologies based on GLSL and CUDA pipelines operating over a multi-pass OpenGL framework.

In the future, our method can be utilized to extend works like Polygonal SH [63, 64] for accommodating SDFs into their framework. But with the advent of works like Instant-NGP and NRC [36, 37], Montecarlo-based rendering techniques are garnering more interest than analytical approximations to the Rendering Equation, nevertheless, most older systems still use SH-based rendering where this work can still be incorporated.

Related Publications

- **Sirikonda Dhawal**, Aakash KT, P.J. Narayanan. *Transfer Textures for Fast Precomputed Radiance Transfer*. In Eurographics 2022 - Posters, https://doi.org/10.2312/egp.20221012
- **Sirikonda Dhawal**, Aakash KT, and P.J. Narayanan. *PRTT: Precomputed Radiance Transfer Textures*. arXiv:2203.12399(2022), https://arxiv.org/abs/2203.12399
- **Sirikonda Dhawal**, Aakash KT, P.J. Narayanan *Real-Time Rendering of Arbitrary Surface Geometries using Learnt Transfer*. In Proceedings of the Thirteenth Indian Conference on Computer Vision, Graphics and Image Processing (2022). Article 39, https://doi.org/10.1145/3571600.3571640

Bibliography

[1] Diede Apers, Petter Edblom, Charles de Rousiers, and Sébastien Hillaire. Interactive Light Map and Irradiance Volume Preview in Frostbite. *Ray Tracing Gems*, 2019. 14

[2] Laurent Belcour, Guofu Xie, Christophe Hery, Mark Meyer, Wojciech Jarosz, and Derek Nowrouzezahrai. Integrating Clipped Spherical Harmonics Expansions. *ACM Trans. Graph.*, 2018. 14

[3] James F. Blinn. Models of light reflection for computer synthesized pictures. ACM SIGGRAPH, 1977. 9, 17

[4] Kenneth Blomqvist, Lionel Ott, Jen Jen Chung, and Roland Siegwart. Baking in the Feature: Accelerating Volumetric Segmentation by Rendering Feature Maps. In *arXiv*, 2022. 16

[5] Mark Boss, Varun Jampani, Raphael Braun, Ce Liu, Jonathan T. Barron, and Hendrik P.A. Lensch. Neural-PIL: Neural Pre-Integrated Lighting for Reflectance Decomposition. In *Adv. Neural Inform. Process. Syst.*, 2021. 16

[6] Chong Bao and Bangbang Yang, Zeng Junyi, Bao Hujun, Zhang Yinda, Cui Zhaopeng, and Zhang Guofeng. NeuMesh: Learning Disentangled Neural Mesh-based Implicit Field for Geometry and Texture Editing. In *Eur. Conf. Comput. Vis.*, 2022. 16

[7] Blender Online Community. *Blender - a 3D modelling and rendering package*. Blender Foundation, Stichting Blender Foundation, Amsterdam, 2018. 24

[8] Thomas Ryan Davies, Derek Nowrouzezahrai, and Alec Jacobson. On the Effectiveness of Weight-Encoded Neural Implicit 3D Shapes. 2021. xi, 39, 43, 45

[9] Dawson-Haggerty et al. trimesh. 34

[10] Sirikonda Dhawal, Aakash KT, and P. J. Narayanan. PRTT: Precomputed Radiance Transfer Textures. In *arXiv*, 2022. 33, 37

[11] Szabolcs Dombi. ModernGL, high performance python bindings for OpenGL 3.3+. Github, 2020. 24, 34

[12] Fizzler. Rabbit SH18. Shadertoy, 2018. 39

[13] Guy Gafni, Justus Thies, Michael Zollhöfer, and Matthias Nießner. Dynamic Neural Radiance Fields for Monocular 4D Facial Avatar Reconstruction. In *IEEE/CVF Conf. Comput. Vis. Pattern Recog.*, 2021. 33

[14] Gene Greger, Peter Shirley, Philip M. Hubbard, and Donald P. Greenberg. The Irradiance Volume. *IEEE Comput. Graph. Appl.*, 1998. 14, 18, 31

[15] Jie Guo, Zijing Zong, Yadong Song, Xihao Fu, Chengzhi Tao, Yanwen Guo, and Ling-Qi Yan. Efficient Light Probes for Real-Time Global Illumination. *ACM Trans. Graph.*, 2022. 15

[16] Peter Hedman, Pratul P. Srinivasan, Ben Mildenhall, Jonathan T. Barron, and Paul Debevec. Baking Neural Radiance Fields for Real-Time View Synthesis. In *IEEE/CVF Int. Conf. Comput. Vis.*, 2021. 16

[17] Eric Heitz, Jonathan Dupuy, Stephen Hill, and David Neubelt. Real-Time Polygonal-Light Shading with Linearly Transformed Cosines. *ACM Trans. Graph.*, 2016. 10

[18] John T Hooker. Volumetric global illumination at Treyarch. *Advances in Real-Time Rendering*, 2016. 14

[19] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 1989. 11, 33

[20] Jinkai Hu, Milo K Yip, Guillermo Elias Alonso, Shihao Gu, Xiangjun Tang, and Xiaogang Jin. Efficient real-time dynamic diffuse global illumination using signed distance fields. *The Visual Computer*, 2021. 14

[21] David S. Immel, Michael F. Cohen, and Donald P. Greenberg. A Radiosity Method for Non-Diffuse Environments. In *ACM SIGGRAPH*, 1986. 8

[22] Michal Iwanicki and Peter-Pike Sloan. Normal Mapping with Low-Frequency Precomputed Visibility. In *ACM SIGGRAPH*, 2009. 14

[23] James T. Kajiya. The Rendering Equation. *ACM SIGGRAPH*, 1986. 8

[24] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, B. Catanzaro, Paul Ivanov, and Ahmed Fasih. PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation. *Parallel Computing*, 2012. 35

[25] Jaroslav Křivánek, Sumanta Pattanaik, and Jiří Žára. Adaptive Mesh Subdivision for Precomputed Radiance Transfer. In *Proceedings of the 20th Spring Conference on Computer Graphics*, SCCG '04, 2004. ix, 14, 15, 18, 33

[26] Jaakko Lehtinen and Jan Kautz. Matrix Radiance Transfer. In *Proceedings of the 2003 I3D*, New York, NY, USA, 2003. 13, 14

[27] Jaakko Lehtinen, Matthias Zwicker, Janne Kontkanen, Emmanuel Turquin, François X. Sillion, and Timo Aila. Meshless Finite Elements for Hierarchical Global Illumination. Technical report, 2007. 31

[28] Jaakko Lehtinen, Matthias Zwicker, Emmanuel Turquin, Janne Kontkanen, Frédo Durand, François X. Sillion, and Timo Aila. A Meshless Hierarchical Representation for Light Transport. In *ACM SIGGRAPH*, 2008. 14

[29] Yue Li, Pablo Wiedemann, and Kenny Mitchell. Deep Precomputed Radiance Transfer for Deformable Objects. In *Proceedings of the 2019 I3D*, 2019. 16

[30] Linjie Lyu, Ayush Tewari, Thomas Leimkuehler, Marc Habermann, and Christian Theobalt. Neural Radiance Transfer Fields for Relightable Novel-view Synthesis with Global Illumination. In *Eur. Conf. Comput. Vis.*, 2022. 15, 16, 18

[31] Andrew L. Maas. Rectifier Nonlinearities Improve Neural Network Acoustic Models. 2013. 33

[32] Zander Majercik, Jean-Philippe Guertin, Derek Nowrouzezahrai, and Morgan McGuire. Dynamic diffuse global illumination with ray-traced irradiance fields. *Journal of Computer Graphics Techniques*, 2019. 14, 15

[33] Morgan McGuire, Mike Mara, Derek Nowrouzezahrai, and David Luebke. Real-time global illumination using precomputed light field probes. In *Proceedings of the 2017 I3D*, 2017. 15

[34] Harrison Lee McKenzie Chapter. Textured Hierarchical Precomputed Radiance Transfer. 2010. 14, 18, 33, 37

[35] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis. In *Eur. Conf. Comput. Vis.*, 2020. 16, 33, 50

[36] Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. Instant Neural Graphics Primitives with a Multiresolution Hash Encoding. *ACM Trans. Graph.*, 2022. 51

[37] Thomas Müller, Fabrice Rousselle, Jan Novák, and Alexander Keller. Real-time Neural Radiance Caching for Path Tracing. *ACM Trans. Graph.*, 2021. 51

[38] Jacob Munkberg, Jon Hasselgren, Tianchang Shen, Jun Gao, Wenzheng Chen, Alex Evans, Thomas Mueller, and Sanja Fidler. Extracting Triangular 3D Models, Materials, and Lighting From Images. In *IEEE/CVF Conf. Comput. Vis. Pattern Recog.*, 2022. 16

[39] Ren Ng, Ravi Ramamoorthi, and Pat Hanrahan. Triple Product Wavelet Integrals for All-Frequency Relighting. *ACM Trans. Graph.*, 2004. x, 11, 14, 17, 18, 35, 36, 37, 51

[40] Derek Nowrouzezahrai, Patricio Simari, and Eugene Fiume. Sparse Zonal Harmonic Factorization for Efficient SH Rotation. *ACM Trans. Graph.*, June 2012. 13

[41] Jacopo Pantaleoni, Luca Fascione, Martin Hill, and Timo Aila. PantaRay: Fast Ray-Traced Occlusion Caching of Massive Scenes. *ACM Trans. Graph.*, 2010. 14

[42] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*. 35

[43] Inigo Quilez. Pixar Mike Monster Inc. Shadertoy, 2013. 39

[44] Nasim Rahaman, Aristide Baratin, Devansh Arpit, Felix Draxler, Min Lin, Fred Hamprecht, Yoshua Bengio, and Aaron Courville. On the Spectral Bias of Neural Networks. In *Proceedings of the 36th International Conference on Machine Learning*, Proceedings of Machine Learning Research. PMLR. 33, 35

[45] Gilles Rainer, Adrien Bousseau, Tobias Ritschel, and George Drettakis. Neural Precomputed Radiance Transfer. *Comput. Graph. Forum (CGF)*, 2022. 15, 16, 18

[46] Ravi Ramamoorthi. *Precomputation-Based Rendering*. NOW Publishers Inc, 2009. 14, 18, 33

[47] Ravi Ramamoorthi and Pat Hanrahan. An Efficient Representation for Irradiance Environment Maps. In *ACM SIGGRAPH*, 2001. 10, 13

[48] Christian Reiser, Songyou Peng, Yiyi Liao, and Andreas Geiger. KiloNeRF: Speeding up Neural Radiance Fields with Thousands of Tiny MLPs. In *IEEE/CVF Int. Conf. Comput. Vis.*, 2021. 16, 33

[49] Peiran Ren, Jinpeng Wang, Minmin Gong, Stephen Lin, Xin Tong, and Baining Guo. Global Illumination with Radiance Regression Functions. *ACM Trans. Graph.*, 2013. 16

[50] Simon Rodriguez, Thomas Leimkühler, Siddhant Prakash, Chris Wyman, Peter Shirley, and George Drettakis. Glossy Probe Reprojection for Interactive Global Illumination. *ACM Trans. Graphics*, 2020. 14

[51] Ari Silvennoinen and Jaakko Lehtinen. Real-time Global Illumination By Precomputed Local Reconstruction from Sparse Radiance Probes. *ACM Trans. Graph.*, 2017. 14

[52] Peter-Pike Sloan. Stupid spherical harmonics (SH) tricks. In *Game developers conference*, 2008. 13, 14

[53] Peter-Pike Sloan, Jan Kautz, and John Snyder. Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. ACM SIGGRAPH, 2002. 10, 11, 13, 17, 18, 20, 23, 30, 31, 33, 50

[54] Peter-Pike J. Sloan, J. Hall, J. Hart, and John M. Snyder. Clustered principal components for precomputed radiance transfer. *ACM SIGGRAPH*, 2003. 13, 37

[55] John Snyder. Code Generation and Factoring for Fast Evaluation of Low-order Spherical Harmonic Products and Squares. Technical report, 2006. 13

[56] Towaki Takikawa, Andrew Glassner, and Morgan McGuire. A Dataset and Explorer for 3D Signed Distance Functions. *Journal of Computer Graphics Techniques (JCGT)*, 2022. 39

[57] Towaki Takikawa, Joey Litalien, Kangxue Yin, Karsten Kreis, Charles Loop, Derek Nowrouzezahrai, Alec Jacobson, Morgan McGuire, and Sanja Fidler. Neural Geometric Level of Detail: Real-time Rendering with Implicit 3D Shapes. In *IEEE/CVF Conf. Comput. Vis. Pattern Recog.*, 2021. 43

[58] Matthew Tancik, Pratul P. Srinivasan, Ben Mildenhall, Sara Fridovich-Keil, Nithin Raghavan, Utkarsh Singhal, Ravi Ramamoorthi, Jonathan T. Barron, and Ren Ng. Fourier Features Let Networks Learn High Frequency Functions in Low Dimensional Domains. *Adv. Neural Inform. Process. Syst.*, 2020. 33

[59] Natalya Tatarchuk. Irradiance volumes for games. In *Game Developer's Conference*, 2005. 14

[60] Greg Turk. *Generating Random Points in Triangles*. Academic Press Professional, Inc., 1990. x, 33, 34

[61] K Vardis, G Papaioannou, and A Gkaravelis. Real-time radiance caching using chrominance compression. *Journal of Computer Graphics Techniques Vol*, 2014. 14

[62] Ingo Wald, Sven Woop, Carsten Benthin, Gregory S. Johnson, and Manfred Ernst. Embree: A Kernel Framework for Efficient CPU Ray Tracing. *ACM Trans. Graph.*, 2014. 24

[63] Jingwen Wang and Ravi Ramamoorthi. Analytic Spherical Harmonic Coefficients for Polygonal Area Lights. *ACM Trans. Graph.*, 2018. 14, 15, 16, 18, 50, 51

[64] Lifan Wu, Guangyan Cai, Shuang Zhao, and Ravi Ramamoorthi. Analytic Spherical Harmonic Gradients for Real-Time Rendering with Many Polygonal Area Lights. *ACM Trans. Graph.*, 2020. 14, 15, 16, 18, 33, 50, 51

[65] Hanggao Xin, Zhiqian Zhou, Di An, Ling-Qi Yan, Kun Xu, Shi-Min Hu, and Shing-Tung Yau. Fast and Accurate Spherical Harmonics Products. *ACM Trans. Graph.*, 2021. 14

[66] Tianhan Xu and Tatsuya Harada. Deforming Radiance Fields with Cages. In *Eur. Conf. Comput. Vis.*, 2022. 16

[67] Xiuming Zhang, Pratul P. Srinivasan, Boyang Deng, Paul Debevec, William T. Freeman, and Jonathan T. Barron. NeRFactor: Neural Factorization of Shape and Reflectance under an Unknown Illumination. *ACM Trans. Graph.*, 2021. 16, 33

[68] Shuaifeng Zhi, Tristan Laidlow, Stefan Leutenegger, and Andrew Davison. In-Place Scene Labelling and Understanding with Implicit Scene Representation. In *IEEE/CVF Int. Conf. Comput. Vis.*, 2021. 16

[69] Kun Zhou, Yaohua Hu, Stephen Lin, Baining Guo, and Heung-Yeung Shum. Precomputed Shadow Fields for Dynamic Scenes. *ACM Trans. Graph.*, 2005. 13