Interactive Generation of Path-Traced Lightmaps

Thomas Roughton

A thesis

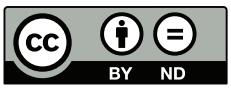
submitted to Victoria University of Wellington

in partial fulfilment of the requirements for the degree of

Master of Science in Computer Graphics

Victoria University of Wellington

2019



Abstract

Indirect illumination is an important part of realistic images, and accurately simulating the complex effects of indirect illumination in real-time applications has long been a challenge for the industry. One popular approach is to use offline precomputed solutions such as *lightmaps* (textures containing the precomputed lighting in a scene) to efficiently approximate these effects. Unfortunately, these offline solutions have historically enforced long iteration times that come at a cost to artist productivity. These solutions have additionally either supported only the low-frequency diffuse component of indirect lighting, yielding poor visual results for glossy or metallic materials, or have used overly expensive approximations.

In recent years, the state of the art lightmap precomputation pipeline has shifted to using highly vectorised path tracing, often on GPU hardware, to compute the indirect illumination effects. The use of path tracing enables *progressive rendering*, wherein an approximation to the full solution is found and then refined as opposed to solving for the final result in a single step. Progressive rendering through path tracing thereby helps to provide rapid iteration for artists.

This thesis describes a system that can progressively path-trace indirect illumination lightmaps on the GPU. Contributing to this system, it introduces a new gather-based method for sample accumulation, enhances algorithms from prior work, and presents a range of encoding methods, including a novel progressive method for non-negative least-squares encoding of spherical basis functions.

In addition, it presents a novel, efficient solution for high-quality precomputed diffuse and low-frequency specular indirect illumination that extends the Ambient Dice family of spherical basis functions. This solution provides comparable or better specular reconstruction to prior work at lower runtime cost and has potential for widespread use in real-time applications.

Acknowledgments

Thank you to my supervisors, Taehyun Rhee and Andrew Chalmers.

Thank you also to Peter-Pike Sloan for his generous help in a number of email conversations around spherical basis functions, to Michał Iwanicki for his help in implementing the Ambient Dice basis, and to Matt Pettineo for his implementation of my progressive least-squares algorithm within The Baking Lab and his invaluable blog series on spherical Gaussians.

The codebase for the core implementation framework was developed in conjunction with Joseph Bennett. Joseph has also been the source of a great deal of interesting conversation over my time at university and an invaluable sounding board for ideas; it would have been hard to stay motivated for this long without him.

Finally, thank you to my parents, extended family, and friends for their support throughout my time at university.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Lightmapping is a long-used technique for caching precomputed indirect lighting, enabling global illumination effects in real-time applications with static geometry and fixed lighting (Figure 1.1). Lightmaps scale well across a wide range of hardware, making them useful for everything from mobile games to high-end PC applications.

Despite this, lightmaps have long faced significant workflow and quality limitations. The generation process for lightmaps (known as 'baking' the lightmaps) generally takes on the order of minutes to hours [1] and is performed in a non-interactive manner, hindering artist productivity as they wait for the result of their work, and the resulting lightmaps poorly represent the glossy specular component of indirect lighting.

Major strides in these areas have been made over the past few years. The precomputation process has been democratised by the use of GPU-based path tracing to precompute the results, enabled by continuing increases in GPU performance and a focus from hardware vendors on ray-tracing acceleration [2]. Prototype solutions for GPU path-tracing lightmappers have been presented in prior industry work within the Unity and EA's Frostbite engines [3, 4], significantly reducing the friction of working with lightmaps by enabling artists to interactively preview the lightmapped results of their 3D scenes.

Unfortunately, building a GPU path-tracer is non-trivial and a path-tracing lightmapper even more so, and the public presentations of prior works have contained only fairly high-level details. While more detailed resources have begun to emerge near to the time of publication [1], comprehensive system overviews remain lacking.

For reconstructing specular BRDFs, prior coarse approximations such as an Ambient and Highlight

Figure 1.1: Indirect lighting from lightmaps in the Sponza Atrium scene [5].

(a) Real-time render using indirect illumination from a baked lightmap. The lightmap uses the Ambient Dice cosine-lobe variant for both diffuse and specular (Section 8.3).

(b) Path traced reference.

Direction term [6, 7] or extracting the primary specular direction from L1 spherical harmonics [8] have been supplemented with new sets of spherical basis functions; notably, spherical Gaussians [9] were applied to the context of lightmapping by Pettineo and Neubelt [10] in 2015. While spherical Gaussians enable reasonably accurate low-frequency specular lighting reconstruction, they are expensive to evaluate at runtime and are difficult to encode during precomputation.

1.1 Thesis Aims and Contributions

In this thesis, I aim to achieve three core goals:

- To detail a comprehensive architecture for building a progressive path-tracing GPU lightmapper, which by its nature enables fast, iterative artist workflows.
- To accelerate the lightmapper, delivering high quality previews and results in less time.
- To improve the visual quality of the produced lightmaps, focusing particularly on specular reconstruction.

These three core aims – informally to *make it work*, to *make it fast*, and to *make it look good* – are achieved by way of many independent contributions, some of which advance multiple of the goals; a fast system is not independent of the architecture of the system, and likewise the architecture must be designed for the visuals it is intended to produce. However, if each contribution is categorised by its primary goal:

- Pursuing the goal of detailing a comprehensive architecture, this thesis:
  - Summarises the underlying split-kernel architecture [11] that is state-of-the-art for performant GPU path tracers.
  - Describes how adaptive rendering may be implemented in the context of a split-kernel GPU path tracer and how those adaptive rendering capabilities may be utilised in a lightmap path tracer.
  - Provides a detailed overview of the lightmap path tracing renderer, which is responsible for computing indirect lighting and encoding it into a lightmap texture.

  - Provides in-depth implementation details for a prior algorithm [8] for packing parameterised meshes into a lightmap atlas.
  - Describes how the result of the parameterisation algorithm may be compressed and used to generate samples on the GPU.
  - Provides a detailed derivation of the mathematical formalism behind spherical basis functions.

- Pursuing the goal of accelerating the lightmapping process, this thesis:

  - Extends the adaptive rendering techniques to perform camera-based lightmap sampling where lightmap samples are prioritised on the user's current view, providing faster visualisation.
  - Details how SIMD instructions recently exposed on GPUs can be used to accelerate the implementation of a GPU-based radix sort, which in turn can be used to accelerate other parts of the path tracing process.
  - Evaluates the importance of ray direction sorting to provide coherent workloads for GPU path tracing.
  - Describes how a tile-based rather than row-based indexing scheme can provide performance gains for path tracing, and gives equations mapping between tile-based and row-based indexing.
  - Overviews performance-quality tradeoffs for the path tracing estimator enabled by integration with an existing rasteriser such as biased light sampling and irradiance caching.

- Pursuing the goal of improving the visual quality of the produced lightmaps, this thesis:

  - Details a novel algorithm to efficiently generate a prefix-sum offset buffer that maps from pixel indices into indices within the paths buffer, enabling gathering of all paths affecting a given pixel.
  - Shows how an offset buffer may be used to perform filtered and multi-step sample accumulation, improving image quality and enabling a class of progressive lightmap encoding techniques not otherwise possible.

- Contributes a novel method for progressive least-squares encoding that supports approximate non-negative solves, enabling a wide range of arbitrary spherical basis functions for use in progressive lightmap encoding.
- Presents a novel method for unified diffuse and specular reconstruction derived from the Ambient Dice [12] family of basis functions, achieving comparable specular and improved diffuse reconstruction at as little as half the runtime cost.

1.2 Thesis Structure

This thesis is divided into three main sections, loosely corresponding to the aims outlined above. The first section, comprising of **Chapters 3** and **4**, focuses on how the lightmapper can be made to function; it covers the core components necessary to build a GPU-based path tracing lightmapper and introduces a framework into which later techniques can be composed. The second section, comprising of **Chapters 5** and **6**, is focused on how the lightmapper can be made to converge quickly and provide fast iteration. The final section, comprising of **Chapters 7** and **8**, looks at how lighting information can be stored in a manner that allows arbitrary normal directions and reconstruction of both diffuse and specular lighting, producing more visually appealing final results.

Additionally, background material is contained in **Chapter 2**, and the appendices provide extra system details. The sum total of this material is a system that can produce and render lightmapped indirect diffuse and specular lighting as seen in Figure 1.1.

Many independent techniques will be evaluated over the course of this thesis, and as such results (including performance statistics, images, and error metrics) will either be contained inline in the chapters or can be found in the appendices. It is recommended that images be viewed in digital format for best resolution and colour accuracy.

To overview the contents of each chapter in more depth:

- **Chapter 2** provides a background to this work. It includes a general introduction to lightmaps, summarises the related work, discusses different possible techniques for lightmap baking, and justifies the choice of path tracing with reference to the state-of-the-art in GPU hardware and path tracing research. Additionally, it provides an introduction to GPU SIMD architectures, summarises wave-

front path tracing as a method of formulating path tracing for the GPU, and outlines GPU radix sort.

- **Chapter 3** provides a basic framework for the lightmap path tracing renderer. It then introduces a filtered accumulation method, an adaptive rendering method, alterations to support lightmap baking, and finally a method of implementing camera-based lightmap sampling.

- **Chapter 4** describes practical considerations of the method used within EA's Frostbite [8] for parameterising geometry into lightmaps, including performance metrics for different implementations and a simple means of compression necessary for copying the parameterisation for use in GPU path tracing.

- **Chapter 5** describes methods of accelerating the path tracer by improving coherence. In particular, it discusses how stream compaction can be integrated into the path tracing framework, describes how GPU SIMD operations can accelerate radix sort on the GPU, evaluates ray direction sorting as an acceleration method, and introduces a tile-based indexing method that helps to achieve coherence by grouping similar rays.

- **Chapter 6** describes methods of accelerating the path tracer by reducing variance. It overviews next event estimation, progressive sample sequences, and importance sampling as methods of variance reduction. It also discusses how biased light sampling or irradiance caching can more quickly produce an image at the cost of bias.

- **Chapter 7** delves into spherical basis functions as a method for encoding radiance or irradiance, providing the mathematical formalism for least-squares encoding of any general set of linear basis functions. That formalism is then extended with a novel method for progressive least-squares and non-negative least-squares encoding of spherical basis functions in an efficient and simple-to-implement manner.

- **Chapter 8** overviews the spherical harmonic, spherical Gaussian, and Ambient Dice families of basis functions. For Ambient Dice, the cosine-lobe variant is extended to provide diffuse and specular reconstruction from encoded radiance lobes. This reconstruction is then compared with spherical Gaussians and spherical harmonics, demonstrating its applicability as an efficient encoding format for low-frequency lighting.

Appendix A briefly discusses how the path tracer was validated against the rasteriser and the open source PBRT renderer [13].

Appendix B gives an overview of the custom 3D engine *LlamaEngine* and the *SwiftFrameGraph* rendering framework [14], which serve as the base implementation frameworks for this thesis. It also includes technical details pertaining to integrating the RadeonRays [15] and Metal Performance Shaders ray-tracing frameworks into the pre-existing engine.

Appendix C provides a detailed image comparison of the Ambient Dice and spherical Gaussian encoding techniques and reconstruction quality from each on a range of environment maps.

Appendix D provides images of a number of test scenes referenced within this thesis.

1.3 Test Hardware

Since results and performance numbers are interspersed throughout each chapter, it would be cumbersome to restate the test hardware alongside every result. Therefore, unless otherwise stated, all performance numbers in this thesis are derived from a computer with the following specifications:

- **Operating System:** macOS 10.14 Mojave
- **CPU:** Intel Xeon W 2150-B (3.0 GHz base clock, 4.5GHz Turbo Boost, 10 physical cores, 20 logical cores)
- **GPU:** AMD Radeon Pro Vega 64 (11TFlops at single precision, 16GB HBM2 at 1.57Gbps)
- **RAM:** 64 GB 2666 MHz DDR4

This hardware serves as a reasonable baseline for a high-end artist workstation. In particular, the GPU architecture (AMD's Vega) is expected to be highly similar to the AMD architecture in next generation consoles [16], and thus its performance characteristics are a good indicator for future mainstream hardware.

Chapter 2

Background

2.1 An Overview of Lightmapping

We have an innate understanding of how light and materials interact to shade the world that we see. We know where the sun is in the sky from where a shadow lies on the ground; we can tell whether a surface is soft or hard or whether it will be cool or warm to the touch on a particular day from only a glance. We understand how light bounces around spaces: that a white wall will lighten a room, a black wall will darken it, and an orange wall will bathe everything in a warm hue. The field of computer graphics has another understanding: the set of phenomena underlying these interactions are incredibly complex, multifaceted, and therefore difficult to accurately and efficiently simulate.

In interactive media, we often want to recreate these effects so that the people interacting with these media feel immersed. The realism we can achieve in real-time has increased over time due to advances in both computing power and research, but even today there are a number of phenomena that we are unable to accurately reproduce or that are too expensive to perform on most consumer hardware.

Real-time media imposes a very restrictive time budget, wherein a new image must usually be rendered in between around 16 and 33 milliseconds each frame. Thankfully, there are far looser time restrictions in the production stages of these media, and we can use that time to precompute and cache many of the calculations necessary. For example, as one solution, we can simply look up the lighting for a scene in a precomputed table or texture. If the camera position is fixed, this simply amounts to rendering a movie ahead-of-time; however, most real-time applications target interactivity, where the user is able to navigate the world of their own accord.

A straightforward method to achieve this is to render out a series of pictures or animations of the scene from different perspectives. This was a commonly used technique in early point-and-click games; a particularly prominent example is the 1993 game *Myst* [17], which allowed players to explore a 3D world from a series of predefined viewpoints. Today, however, users expect to be able to move and look around freely. That presents a different challenge: caching the lighting in a *view-independent* manner.

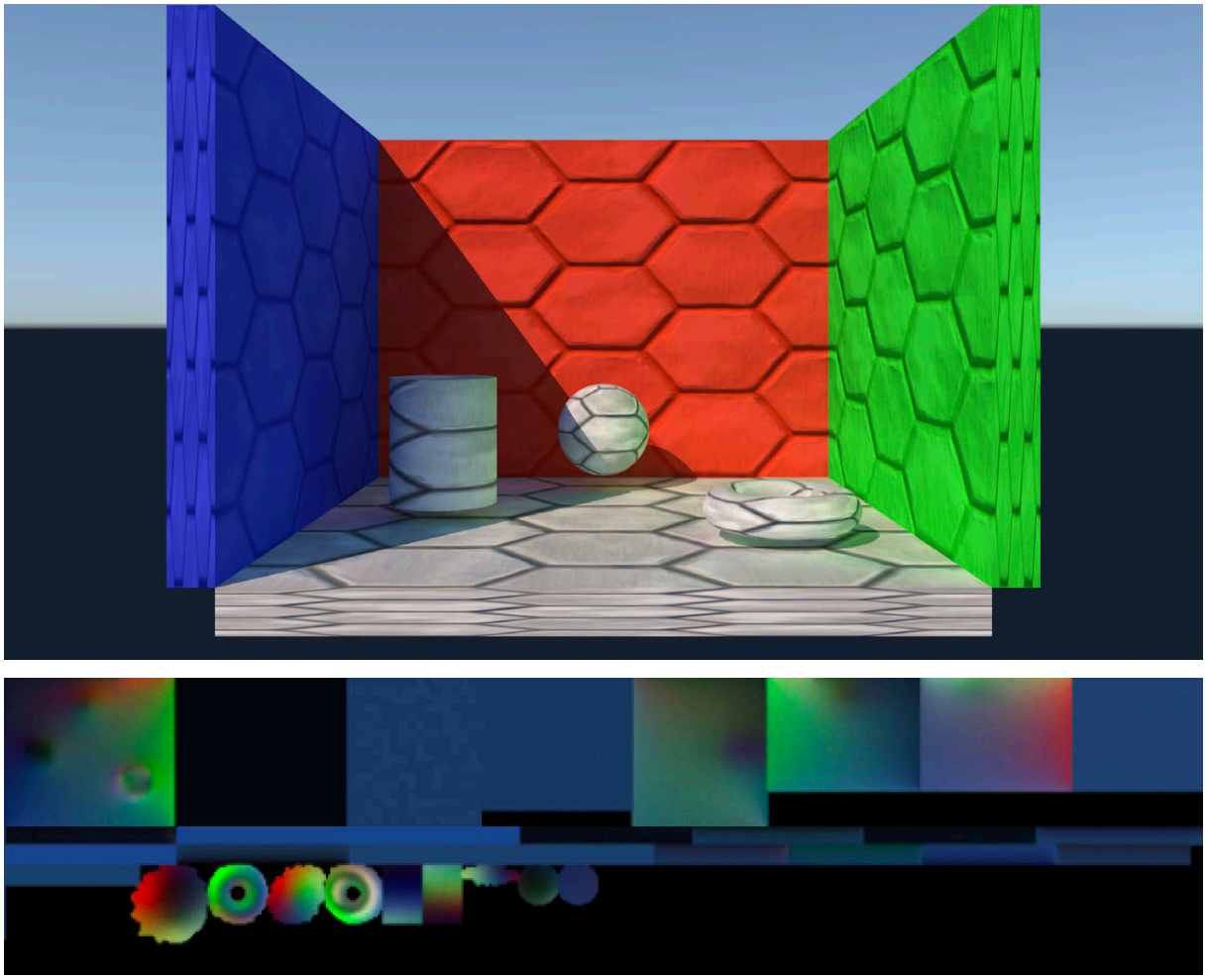
One method of doing so is called *lightmapping*. A *lightmap* is a texture that can be queried for the incident *irradiance* at points on surfaces within a scene. Mapping from locations on surfaces in the scene to locations within the lightmap texture requires a *lightmap parameterisation* for the scene. If we assume that each surface is comprised of a number of triangles (as is commonly the case), the lightmap parameterisation defines the texture coordinate (a two-dimensional value also known as a *UV*) corresponding to each vertex of each triangle. When we render each triangle to the screen, we can look up the texture coordinates for each of its vertices, blend between them depending on where we are on the triangle, and then sample the lightmap to get the irradiance at that location.[1] An example of a scene lit using a lightmap and its corresponding lightmap texture is given in Figure 2.1.

Note that lightmaps cannot store the irradiance at every point: irradiance is continuous over a surface, and textures are discrete, storing data within each texel. However, by blending between neighbouring texels we can get an approximation for the irradiance at every point, with the quality of the approximation being determined by the footprint of each lightmap texel on the screen; this in turn is determined by the density of the texels (i.e. the ratio of texels to world-space units), the viewing perspective, and the distance from the viewer to the object. If we assume that the outgoing radiance (the light from a point on a surface in a particular direction) is the same over all directions on a hemisphere – an assumption that holds true for a particular model of material reflectance called *Lambertian reflectance* – an infinitely high resolution lightmap could give us an exact view of the lighting within a scene from any arbitrary viewpoint.

In reality, lightmaps are of limited resolution; in fact, in early uses you might have had only one lightmap texel for a one-metre square area [19]. We would prefer to have detailed colour textures for different objects within the scene; a single colour for each metre is in most contexts highly undesirable. Fortunately, we can begin to separate out information from the lightmap. For example, diffuse illumination using Lambertian reflectance is defined as:

[1] An alternative approach to lightmapping is to store the radiance data per-vertex directly in the mesh. This presents alternative trade-offs: it bypasses the need for a unique parameterisation of the scene, but does tie the number of radiance samples to the triangle density of the mesh and prohibit instancing.

Figure 2.1: The diffuse Baking Lab scene and its corresponding scalar indirect illumination lightmap. [18]



$$I = \frac{\rho}{\pi} \int_{\Omega} R(\omega) \cdot (\omega \cdot \vec{n}) \ \mathrm{d}\omega \tag{2.1}$$

where I is the outgoing irradiance, ρ is the albedo (diffuse colour), $\vec{n}$ is the surface normal, and $R(\omega)$ gives the incoming radiance in direction ω. Since the albedo is separate from the radiance integral, we can store them separately; for example, we can choose to store only the incident radiance over the hemisphere in the lightmap and then later multiply by the surface albedo. Using this approach, we can combine highly detailed albedo textures with low-frequency lightmaps to achieve realistic results.

This general method was first used by id Software's 1996 game *Quake* [20] (Figure 2.2) for all lighting information and works fairly well. Hardware and rendering have significantly advanced in the years since, however; we can now trivially evaluate many different light types in real-time, and we now make use of many different physically-based material reflectance models.[2] This ability to evaluate complex lighting in real-time would seemingly obviate the need for lightmaps.

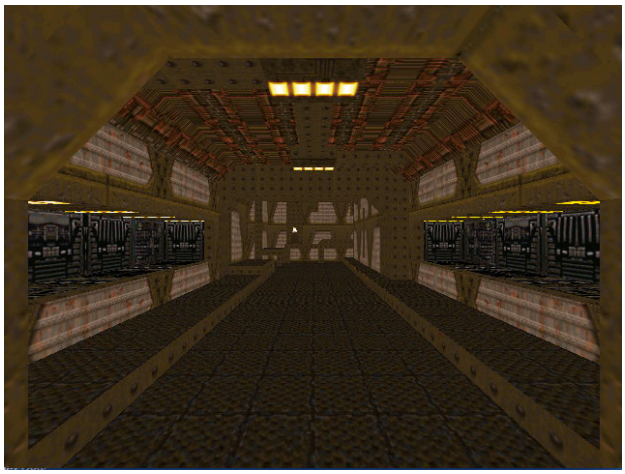
There remain, however, a number of phenomena that are prohibitively expensive to accurately simulate in real-time. The most prominent of these is global, or indirect, illumination. Global illumination accounts for the paths light takes from surface to surface; the aforementioned warm glow from an orange wall is one example of this, wherein light from a source such as the sun hits the wall and then scatters light out into the remainder of the room; traditional lighting, conversely, only accounts for the first bounce of a light onto scene surfaces.

There are many techniques to approximate indirect illumination in real-time applications, varying in accuracy and performance profile. Today, lightmaps remain one of the most inexpensive options, providing high quality at low computational cost; for diffuse irradiance, this cost is a single texture lookup per pixel. To store indirect illumination in a lightmap, each texel needs to contain all of the incident radiance at a point that did not directly come from a direct light source – in other words, it should contain all of the radiance that arrived at the surface by being bounced off another surface.

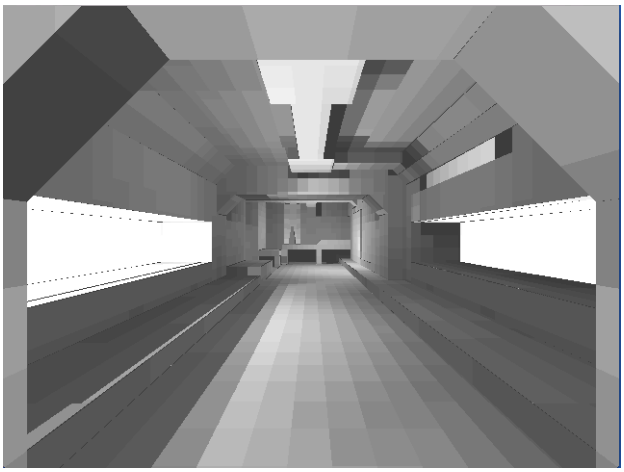
Indirect illumination lightmaps are used in many modern applications. They are particularly invaluable when targeting high frame-rates (such as competitive action games or virtual reality) or low-end or power-constrained hardware (such as smartphones or tablets), but are also broadly useful in providing a complex effect in an inexpensive way; many high-end console games targeting moderate framerates have also incorporated lightmaps [22, 23].

[2] Material reflectance models are often abbreviated as *BRDF*s, or bidirectional reflectance distribution functions. BRDFs are the subset of *BSDF*s, or bidirectional scattering distribution functions, that do not include transmission.

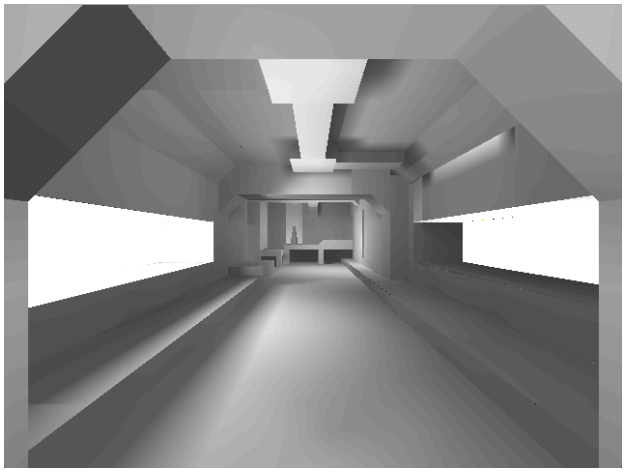
Figure 2.2: Lightmaps in id Software's 1996 game Quake [20]. Image credit Bush 2015 [21].



(a) Textures only



(b) Point-sampled lightmap



(c) Bilinearly-sampled lightmap

(d) Lit scene

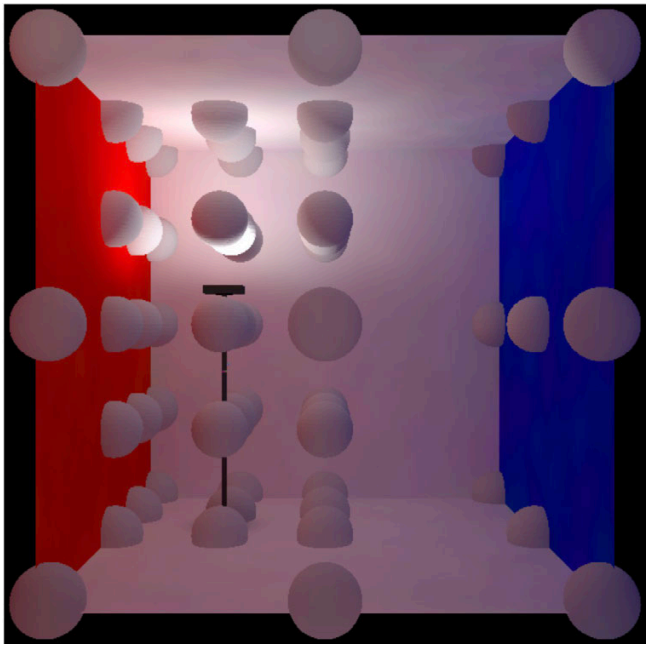
2.1.1 Irradiance Volumes

There are other noteworthy methods of storing pre-computed lighting than textures mapped to scene surfaces. One such method is the use of *Irradiance Volumes* (Figure 2.3) [24] (introduced to the context of real-time applications by Tatarchuk in 2005 [25]): 3D data structures that can be queried for the irradiance (or some other precomputed lighting information) at continuous locations in 3D space. A simple representation of an irradiance volume might be a grid, where the lighting can be interpolated between cells in that grid.

Irradiance volumes are useful in conjunction with lightmaps due to their ability to provide approximate indirect illumination for dynamic objects (whose position was not known at lightmap generation time) and volumetric materials.

Many of the same considerations for lightmaps apply to irradiance volumes, and the process for generating them is much the same. There are two main distinctions: one, a lightmap stores lighting information in a hemisphere, whereas an irradiance volume stores lighting in all directions, and two; the parameterisation of an irradiance volume is usually much simpler, often corresponding to 3D world-space positions. With the exception of those two specific areas, however, almost all of the techniques and methods discussed in this thesis are equally applicable to both lightmaps and irradiance volumes.

Figure 2.3: An irradiance volume. Each sphere shows the incident lighting at a particular point in the scene. Image credit Greger 1996 [24] (Figure 4.16).



2.2 Augmenting Lightmaps

We can store more than simple colour values – such as the cosine-weighted Lambertian irradiance from Equation 2.1 – within lightmap textures. There are two main reasons why we would want to do so: normal mapping and non-Lambertian BRDFs.

In rendering, we often want to give the illusion that a mesh has more detail than it actually has, and one method to do so is called *normal mapping* (Cohen et. al. 1998) [26]. Normal mapping adjusts, per-pixel, the interpolated shading normal based on a high-resolution texture; it is what allows a two-triangle plane to look convincingly like a brick wall. It presents a problem for irradiance caching, however; when we precompute, or *bake*, a lightmap, we do so assuming a particular surface normal. If that surface normal differs at runtime (whether due to high-detail normal maps or the use of lower-resolution proxy geometry for baking) then the lighting information will be incorrect.

Non-Lambertian BRDFs present another challenge: view-dependency. In general, BRDFs are parameterised by both the incoming radiance direction (as in Lambertian reflectance) and by the outgoing view direction. This is what makes materials appear reflective or shiny; the lighting changes as you move around the object. All real-life materials have some degree of view-dependency, with the extent depending upon the material. If we assume that the outgoing radiance is constant with respect to viewing angle we heavily limit the types of materials and scenes we can recreate.

Fortunately, there are ways of storing the radiance that allow us to retain some directionality. For example, spherical harmonics (Section 8.1), which were first introduced into the context of rendering by Ramamoorthi and Hanrahan in 2001 [27], are a method of representing spherical functions in frequency space. Given a spherical harmonic function representation of radiance, it is possible to query that function for the radiance in any particular direction.

Conveniently, spherical harmonics also allow us to analytically and cheaply compute the Lambertian *irradiance* in any direction: the integral in Equation 2.1 becomes a simple dot product (Section 7.4). When combined with normal maps, the spherical harmonic can be evaluated in the direction from the normal map to achieve indirect lighting that interacts with the bumps in the surface.

Irradiance from diffuse materials is inherently low-frequency – the material 'blurs' together radiance from over a hemisphere – which means that in frequency space we only need to consider the first few spherical harmonic bands.[3] For lightmaps, only the L0 and L1 bands (comprising of four coefficients per

[3] Spherical harmonics consist of a series of bands, each of which represents an increasingly higher-frequency component of

colour channel in total) are commonly used [8].

Representing indirect specular – the high-detail reflections that shift based on the viewing direction – is substantially trickier. The core issue is that specularity for smooth materials is inherently *high*-frequency; given a mirror (a perfectly smooth, purely specular surface), you see the perfect reflection of whatever is in the mirror's reflection direction relative to your viewing position. If, however, we restrict ourselves to rough materials, we have some better options.

One approach, used by e.g. EA's Frostbite engine [8], is to treat the spherical harmonic as a directional or area light. The three coefficients of the L1 band of a spherical harmonic represent the average direction of the light, and the length of that direction vector indicates *how* directional the light is; larger L1 band coefficients mean that the light is fairly focused rather than spread. This can then be used to adjust the intensity of the specular highlight. While often visually plausible, this is also very inaccurate, and can have distracting artefacts when the average direction varies over a surface. Consider the case of a glossy plane with an area light source at its top and right; you would expect two specular highlights, but instead the specular highlight would appear to turn a corner on the surface as the primary direction transitioned from one light to the other.

A similar approach is employed by Sloan and Silvennoinen (2018) [6], who propose solve and compression methods for the Ambient and Highlight Direction (AHD) encoding method, which was first applied in id Software's Quake 3 (1999) [28] and independently extended for use in lightmaps by Lazarov (2011) [29] and Iwanicki (2013) [22]. AHD is an inexpensive representation analogous to L1 spherical harmonics but with a different method of evaluation, and is better suited to lightmaps than spherical harmonics due to its restriction to the hemisphere; since all lighting information for lightmaps is contained within the hemisphere surrounding the surface normal, encoding methods such as spherical harmonics which encode spherical information are comparatively wasteful. When performance is the top priority, AHD currently presents the best relative quality [6].

Another alternative for specular BRDFs, introduced by Chen and Liu at SIGGRAPH 2008 [7] and used in Bungie's 2007 game *Halo 3*, is to project the BRDF onto a spherical harmonic basis and then integrate it with the indirect radiance. In frequency space, this integration can be cheaply computed as a vector dot product. Since specular lighting is higher frequency than diffuse this requires including the five higher-frequency L2 band spherical harmonic coefficients in the reconstruction, bringing the total number of coefficients up to nine per colour channel. This approach also necessitates the use of

the signal.

precomputed lookup tables for the BRDF coefficients, which in turn means that the BRDFs are restricted to have an isotropic response.[4] Real-world materials have anisotropic specular response: the highlights stretch out at grazing viewing angles.

One option would be to use higher-dimensional lookup tables to enable anisotropic specular responses. However, that brings us to a larger issue with this technique. Low-band spherical harmonics are simply too low-frequency to accurately represent high-frequency, high dynamic range indirect information. To achieve higher quality, we need to keep on increasing the amount of data, but the quantities soon become untenable; the L3 band adds seven coefficients per channel and the L4 band adds another nine. For real-time performance, the bandwidth cost imposed by simply fetching all that data per-texel is substantial.

Spherical Gaussians are a more recent alternative that can provide results with diffuse irradiance reconstruction error up to somewhere between L1 and L2 spherical harmonics. They were first introduced by Wang et. al. in 2007 [30], and have since been most prominently used in Ready at Dawn's 2015 game *The Order: 1886* [31]. Spherical Gaussians are particularly interesting because of their ability to represent specular light sources, thereby providing a more accurate approximation to indirect specular. This approximation only works well for rough specular materials, however; highly reflective materials require very high frequency lighting information, which implies a large number of coefficients and a storage and bandwidth cost that is usually untenable for real-time applications, regardless of encoding format.

Spherical Gaussians are radial basis functions, which, in the context of spherical functions, means the value depends on the angle between some spherical direction and the query direction. When using linear bases for lighting, each basis function can be treated as an independent light source and used to evaluate irradiance. If the linear basis is comprised of a single type of radial basis function oriented in different directions, the same equations or fits may be used to evaluate irradiance for each component basis function; this is an important advantage of radial basis functions over spherical harmonics, for which each band has a different response and therefore requires different fits or lookup tables.

In Section 8.3, I introduce a new method for reconstructing low-frequency specular using a radial basis function from Sloan and Iwanicki's 2018 paper *Ambient Dice* [12]. This method offers comparable specular reconstruction quality and better diffuse reconstruction than spherical Gaussians at lower computational cost.

[4] Most BRDFs are parameterised by some roughness parameter for each perpendicular tangent direction, the view direction, and the reflection direction. This is generally too many dimensions to store in a lookup table; instead, common approximations are to assume that the view direction is aligned to the surface normal and that the roughness parameter is the same for both tangent directions.

2.3 Precomputing the Indirect Lighting

Given that an application will use precomputed lighting information stored in lightmaps or irradiance volumes, the next question is how that lighting information should be precomputed. When considering different techniques, we must have some criteria to evaluate each by; image quality is one aspect; time to complete the precomputation process is another; but arguably the most important is the time taken before the artists have a preview of the final result.

Consider an artist working on a level in a 3D game. They have just finished placing lights around the scene, carefully shrouding certain objects – collectables or secrets, perhaps – in murky darkness while subtly illuminating the path forward for the player to take. Satisfied with their work, they submit their level to a job queue to bake the indirect lighting.

They return to the level a number of hours later when the bake process is complete. Aghast, they find that their carefully designed lighting has been muddied, and their gameplay cues hidden. Their carefully shadowed corner is now filled out by light bounced off the brightly lit main path, and that main path is no longer nearly as distinct, the contrast with the surrounding area lost. They make further adjustments to the lights, compensating for the effects of the indirect lighting output by the bake process, and submit their level to the queue, preparing to wait a number of hours more with no guarantee of a better result.

The iteration time imposed by this workflow is highly undesirable. As an artist, you want to be able to quickly preview the results of your changes. The preview does not need to be production quality – minor visual artefacts are fine – but it does need to give an impression of the entire scene. The goal, therefore, is a workflow where the result is *progressively rendered*. Ideally, it should also be possible to look around and interact with the scene while the bake process is ongoing, with changes to either the geometry or lighting restarting the bake process. A progressively rendered, interactive workflow will therefore be our main criterion as we overview a few possible methods: recursive rasterisation, radiosity, and path tracing.

2.3.1 Recursive Rasterisation

One candidate technique is called *recursive rasterisation* [23]. Given an understanding of a standard rasterisation pipeline, the concept is fairly simple. Firstly, note that single-bounce indirect lighting is simply direct lighting as seen by another surface multiplied by that surface's BRDF; in effect, every illuminated scene surface itself becomes a light source. Given we have the ability to rasterise the direct lighting in the

scene from some arbitrary viewpoint – the arbitrary viewpoint usually being the main camera – we should therefore be able to render the scene from the viewpoint of each texel in the lightmap. That, integrated with the BRDF, then becomes the first-bounce indirect lighting that is stored in the lightmap.

We can recursively repeat this process to however many bounces is desired, with each render adding another bounce of light. Note that in the absence of specular light transport (i.e. with only diffuse BRDFs) and with non-white surface albedos, successive bounces will contribute progressively smaller fractions to the result, and at some point the difference will be imperceptible.

The main benefit of this approach is that it is trivial to implement within a standard rasteriser; material and lighting models automatically match and are consistent between direct and indirect lighting. However, it has a few major downsides. Firstly, every texel must be rendered in succession;[5] it is an inherently serial process, meaning that each texel's contribution for a particular bounce must be calculated in turn before subsequent texels are computed. Secondly, we incur a CPU overhead for generating and submitting the draw calls for each texel. Finally, we are also limited by all the issues of rasterisation: the result will be limited in resolution and aliased (so crucial high-frequency lighting could be missed), phenomena such as light refracting through other objects or translucency is difficult to represent correctly, and the rasterised image will be distorted by the projection used.[6]

Recursive rasterisation renders somewhat progressively – texels can be filled in in turn, and the full lightmap can be previewed after each bounce –, but the iteration time is lengthy. Despite this, it has been successfully used in production for applications such as *The Witness* (Thekla Inc., 2016) [23].

2.3.2 Radiosity

Radiosity is a method for computing the diffuse interreflection between surfaces in a scene. First adapted to the field of computer graphics by Goral et. al in 1984 [33], radiosity methods involve parameterising the scene into a set of *patches* (akin to lightmap texels), computing a *visibility factor* between each pair of patches, and then solving for the irradiance at each patch by iteratively adding the visibility-factor-weighted irradiance at all other patches within the scene.

[5] It is possible to render multiple texels at once if multi-view rasterisation is supported in the graphics hardware, depending upon the limits of the hardware and the projection used. In a best-case scenario on current hardware up to 16 texels could be rendered at once using low-quality paraboloid projection mapping [32].

[6] In rasterisation, only linear projections can be used – linear here meaning that every triangle in the source space maps to a plane in the final projected space and cannot be warped or curved. The projection we want – spherical projection over a hemisphere, where every triangle becomes slightly curved – is nonlinear, and so will either produce artefacts or must be approximated with perspective frusta such as square frusta on the faces of a cube.

Radiosity produces robust results and is fairly simple. However, it has a few major limitations, even within the constraints of solely diffuse interreflection. One is that every patch must evaluate the contribution of every other patch (or, at the least, every other mutually visible patch), which causes the technique to scale poorly with resolution. Another is that shadowing is poorly resolved with low patch density since lighting is decoupled from visibility: there is an assumption that incident illumination is constant over a patch, which is not usually the case.

The more major issue is that radiosity depends upon the ability to compute the visibility factors, which in turn relies upon either rasterisation or the ability to trace rays between the patches.[7] This is largely unimportant when radiosity is used for runtime relighting of a scene: the visibility factors can be precomputed offline and then the radiosity algorithm can be applied at runtime (with all of its associated costs). However, when all lighting is precomputed, the same process used to compute the visibility can also be applied iteratively or recursively to compute the lighting directly, which reduces the usefulness of the radiosity algorithm in these scenarios: it is simpler to recursively apply an already-implemented algorithm than to implement a separate method which itself must be iteratively applied.

2.3.3 Path Tracing

Path tracing is an unbiased, probabilistic method to estimate radiance by casting rays within a scene. The idea is that we can estimate the radiance or irradiance at any point within the scene by simulating the paths that light would travel, accounting for how the light is scattered or absorbed by surfaces and media along the way. In recent years, path tracing has become the standard for offline rendering; Pixar's RenderMan recently shifted to path tracing as its primary algorithm [35], and Disney's Hyperion [36], Weta Digital's Manuka [37], Dreamworks' MoonRay [38], and many other production renderers were designed with path tracing in mind.

Path tracing at its most basic is a very simple algorithm. From the point at which you wish to estimate the radiance, fire a ray in the direction you are interested in and find where it intersects with the scene. This ray constitutes the first part of a new *path* in the scene, where each path carries some accumulated radiance. At the intersection point, add any emission from the surface to the accumulated radiance along the path (where the added radiance is multiplied by the path's throughput). Then, select a new random direction, multiply the throughput of the path by the surface's BSDF given that outgoing and incoming

[7] To give an example, hemicube rasterisation is often used to compute the radiosity form factor, storing identifiers for the visible geometry rather than radiance values [34].

direction, and fire a new ray into the scene. This process recurses infinitely until either the throughput along the path is zero or a ray escapes the scene. By averaging the radiance accumulated across all paths, the true value of the radiance from the source point in the target direction can be estimated.

This process is an example of Monte Carlo integration [39], wherein an estimate for the true value of a function is obtained by averaging together multiple samples. Initially, the estimate will be very noisy; however, as the number of samples increases, the *variance* will decrease; taking four times the number of samples will halve the variance. Techniques such as importance sampling (described in Section 6.3) can be used to improve the convergence rate of the path tracing process; finding methods to improve the convergence rate for path tracing is an active field of research.

To use path tracing in baking a lightmap, you generate paths originating at each lightmap texel's world space position in a hemisphere around each texel's corresponding surface normal. The representation of the radiance to be stored in the lightmap varies; in the most simple scenario, when Lambertian irradiance (a single colour per texel) is being baked into the lightmap, each path's radiance is added with a weight corresponding to cosine of the angle between the ray and the surface.

Because path tracing is a progressive process, the result gradually converges as more samples are taken. This makes it ideal for the use case of lightmap generation: artists can receive visual feedback on the indirect lighting in their scenes almost immediately, enabling a fast iteration time. In addition, path tracing is more flexible than recursive rasterisation and radiosity, producing accurate results for a range of phenomena that the other methods can only approximate (Figure 2.4).

2.4 GPU Path Tracing

In recent years there has been an ever-increasing interest in using path tracing, which has traditionally been implemented on the CPU in a recursive manner, on GPUs. As many as fourteen years ago, Purcell et. al. hypothesised about the direction future programmable graphics hardware (as opposed to the fixed-function hardware prevalent at that time) might take in *Ray Tracing on Programmable Graphics Hardware* [41]. Although highly hypothetical at the time, their design ended up being remarkably prescient; in particular, their use of a multi-pass architecture helps to achieve better performance through coherency.

The first widely-used GPU path tracing framework was NVIDIA's OptiX (Parker et. al. 2010) [42].

Figure 2.4: A path-traced scene featuring complex reflection, refraction, and area light effects, rendered on the GPU in LlamaEngine (Appendix B.1). Scene credit Mareck [40].

OptiX combined prior work on GPU-optimised ray traversal acceleration structures with a formal API and extensible model, enabling a wide range of algorithms to be implemented. NVIDIA has continued to extend OptiX over the years, and it today serves as one of their three supported APIs for ray tracing on their hardware.

In 2015, AMD first previewed RadeonRays (then AMD FireRays), their OpenCL ray-intersection framework [15], and Radeon ProRender, their OpenCL path tracer, [43] which have since both been open--sourced. The path tracing systems described in this thesis are primarily adapted from those open-source codebases.

2018 saw a large increase in industry interest and investment in GPU path tracing. Firstly, NVIDIA's Turing architecture (branded as NVIDIA RTX) was announced [2], bringing hardware acceleration for ray tracing [44]. API support is provided by Microsoft's new DirectX Raytracing API [45], alongside OptiX and extensions for the Vulkan API [46]. Apple also released their own framework for ray-triangle intersection (much like RadeonRays) at WWDC 2018 [47].

Specific to the topic of lightmapping, Hillaire described a GPU implementation of the lightmapper for EA's Frostbite engine on top of DirectX Raytracing at GDC 2018 [4]; similarly, Unity Technologies integrated AMD's RadeonRays library into their progressive lightmapper [3, 48]. More recently, the integration within Frostbite was described in more detail by Apers et. al. in *Ray Tracing Gems* [1], which also holds discussion of a number of other GPU-based path tracing techniques.

Historically, an issue with path tracing on the GPU has been the low amounts of directly-addressable memory; complex scenes (and the acceleration structures necessary to ray-trace those complex scenes) would not fit in video memory. However, recently professional GPUs have begun to approach desktop-level quantities of RAM; at the high end, GPUs such as the Radeon Pro Vega 64 or NVIDIA Quadro RTX 6000 now have upwards of 16GB of attached VRAM.

With the combination of all of these factors, path tracing, and path tracing on the GPU in particular, becomes increasingly practical. Although GPU-based path tracing is likely still a number of years in the future for general consumer hardware, it is ideal for professional and content-creation level solutions such as lightmap generation: the hardware is now fast enough that the process can occur interactively on user machines and the algorithm itself is significantly simpler or has fewer caveats than other alternatives. As such, GPU path tracing will be the core of the remainder of this thesis.

2.4.1 Adapting Path Tracing to the GPU

The natural description of how a path tracer works involves depth-first recursion (Section 2.3.3). Unfortunately, such recursive algorithms are generally poor fits for GPU hardware. To explain why, we need to introduce a few concepts around how GPUs work. The exact terminology varies depending on manufacturer and graphics API; in this document, the conventions followed will be that of Apple's Metal API [49].

At the most fine-grained level we have the concept of a *thread*. A thread on the GPU carries out a series of instructions and has access to its own memory. In the algorithm described above, each thread would operate on a single path.

However, GPUs operate on a SIMD (single instruction, multiple data) programming model. That means that each thread does not operate independently. Instead, threads within groups called *SIMD groups* operate in lockstep, all executing the same instruction on independent data. This has important implications: any divergence in the instructions each thread executes necessitates the instructions executed by *any* thread to be executed by *every* thread.

For example, in a path tracer some paths may terminate before others, whether because a ray escaped and hit the skybox or because a path was probabilistically terminated by Russian roulette.[8] Ideally, the threads operating on those terminated paths would be able to begin a new path to maximise throughput; however, in a SIMD context, those terminated paths are forced to continue executing until the paths for *all* of the threads within the SIMD group have been terminated.

There can be even more severe consequences if the threads branch and diverge in such a way that every thread has a separate set of instructions to execute. Consider a branching `if` statement that checks the thread's index within the SIMD group: if the body of each `if` statement contained a set of separate instructions then the hardware would need to execute the body of every `if` statement for every thread, thus effectively making the code serial. This type of scenario is entirely possible in real use cases: if every path were to hit a different material in one bounce, for example, and each material required different instructions to evaluate it, the effect is as if one thread evaluated every material and all other threads did nothing. To provide some context, SIMD groups are 64 threads wide in AMD's GCN architecture [50]; this type of divergence would therefore yield a 64$\times$ slowdown.

[8] Russian roulette is a method that probabilistically terminates active paths and increases the throughput of survivors to compensate and keep an unbiased result. It can reduce the work done (since there are fewer active paths at each bounce) at the cost of increased variance.

SIMD configurations are primarily used because they are highly efficient to implement in hardware. In addition, they also carry the advantage of being able to efficiently share data between threads. Any thread in a SIMD group can retrieve the value of any other thread within the same group, which can be useful in algorithms such as parallel reduction. However, threads within a SIMD group cannot generally synchronise with threads in a different SIMD group except through order-independent atomic operations or by using multiple dispatch calls. This can be fairly limiting, since the SIMD group width is hardware-determined.

To alleviate this, SIMD groups are contained within larger groups called *threadgroups*. A threadgroup is a group of SIMD groups that can synchronise between each other, for example by copying data to local threadgroup memory and then yielding so that a different SIMD group can execute. There can be at most around 16 or 32 SIMD groups within a threadgroup on AMD's architecture [50], enabling up to 2048 threads to synchronise between each other. Note that there are trade-offs for having large threadgroups, since every thread within a threadgroup must keep its data (e.g. local variables on the stack) in memory at the same time. If there are fewer threads, that means each thread has access to more high-speed threadgroup memory; if there are many threads, local variables can end up spilling out to the much slower device memory [51].

Threadgroups exist within a grid, which can be one-, two-, or three-dimensional. There is a set number of threadgroups within each dispatch call [9], where the number can either be specified directly by the CPU or stored indirectly in a GPU buffer. Threadgroups cannot synchronise with each other within a dispatch call; instead, they must write out intermediate results to device memory for those results to be processed in a separate dispatch.

2.4.2 Wavefront Path Tracing

In 2013, Laine et al. introduced *wavefront path tracing* in *Megakernels Considered Harmful: Wavefront Path Tracing* [11] ('wavefront' being NVIDIA's term for a SIMD group) as a solution to the issues inherent in a depth-first recursive path tracer on the GPU. The core idea is that if the path tracing algorithm is reformulated in a breadth-first manner, the algorithm can then be modularised into multiple, specialised kernels, with potential sorting and compaction steps in between to maximise coherence. Although there

[9] A dispatch call is an invocation of a kernel to be executed on the GPU; to perform a dispatch, the CPU specifies a GPU program, binds parameters for that program, and then dispatches a certain number of threadgroups to execute that program.

are usually differences in implementation between different GPU path tracers, the core structure generally follows the template set out by Laine et. al.

Their approach builds upon earlier methods such as *path regeneration*, first proposed in 2010 by Novák et al. [52]. Path regeneration attempts to maximise the utilisation within SIMD groups by generating new paths to replace terminated paths within each threadgroup. Path regeneration does not address divergent control flow within a SIMD group (e.g. with varying materials or differing depths within the ray traversal data structure) but does ensure that every thread is active and performing useful work.

An alternative approach, and the one primarily used by Laine et. al., is *stream compaction*, first introduced in this context by Wald in 2011 [53]. In short, the buffer of paths is filtered to only include those paths which are active; more concretely, the result of the stream compaction is a buffer containing the indices of the active paths, along with the active path count. In subsequent kernels, each thread either fetches a path using the active indices as an index buffer or, if the thread index within the grid is greater than or equal to the path count, exits early. Stream compaction preserves ordering and therefore coherence between neighbouring paths.

In *Megakernels Considered Harmful*, Laine et. al. also make heavy use of sorting, focusing particularly on materials. Complex BSDFs can be expensive to evaluate; by sorting the ray hits by their materials one can evaluate each material only for the relevant paths, keeping the workload coherent. The decision of whether to split into separate kernels for materials must be made on a case-by-case basis, since there is also an overhead for storing and retrieving the path state in device memory. Laine et. al. found it to be worthwhile for their materials, but decided not to split up the light sampling for different light types into separate kernels; "Light sources that are complex enough to warrant having an individual stage are not as common as complex materials."

In addition to achieving coherence within threadgroups, there is another benefit to having many smaller kernels rather than a single large kernel. Recall that having too many threads in a threadgroup (and therefore too much memory usage in a threadgroup) could cause stack variables to spill from local into device memory (Section 2.4.1). It happens to be the case that the local memory usage of a thread is determined by its *maximum* usage over the course of an entire kernel dispatch; in other words, having one particularly register heavy section of a kernel will limit the rest of the kernel, even if the rest of the kernel does not require as many registers, and therefore having smaller, less register-heavy kernels will improve the occupancy in those kernels. In the words of Laine et. al.:

> To hide (…) latency, GPUs are designed to accommodate many more threads than can be executed in any given clock cycle, so that whenever a group of threads is waiting for a memory request to be served, other threads may be executed. The effectiveness of this mechanism (…) is determined by the threads' resource usage, the most important resource being the number of registers used.

It is worth noting that CPU hardware is becoming increasingly similar to the GPU model. Intel's AVX-512 instruction set, for example, enables operating on 512 bits (or 16 single-precision floats) at once per core [54]. As such, a breadth-first design described here is equally applicable to modern CPUs.

2.5 GPU Radix Sort

Since GPUs perform best on coherent workloads, it is often beneficial to sort the data before a GPU kernel operates on it. For example, in *Megakernels Considered Harmful* [11] Laine et. al. sort the paths to be shaded by material, enabling the subsequent material kernels to avoid evaluating costly divergent branches. As a general rule, sorting will always improve the performance of subsequent kernels; however, whether the performance improvement amortises the cost of the sort varies depending on the workload.

In a GPU-focused environment an efficient *parallel* sort is needed which can distribute the workload over many GPU threads. There are many viable candidates for this – for example, highly optimised variants of merge-sort for the GPU exist [55] –, but one specific type of sort is particularly well-suited to these workloads due to its linear $\mathcal{O}(n)$ rather than $\mathcal{O}(n \log n)$ runtime: radix sort.

GPU radix sort, as outlined by Harada and Howes in *Introduction to GPU Radix Sort* [56], is a three--phase algorithm. The input is assumed to be a buffer of integer keys (and optionally a corresponding buffer of values), with the output being those keys and values in stable sorted order.[10]

Firstly, there is the *count* phase, wherein the number of times each key occurs in the source buffer is counted. This is usually performed by multiple threadgroups, where each threadgroup processes a subrange of the data and outputs the per-threadgroup count for each key to a buffer.

Secondly, the *scan* phase performs a *parallel prefix sum* on those counts. The purpose of the scan phase is to compute the sum of the counts for each key from all preceding threadgroups. As an example, if there

[10] A stable ordering means values with equal keys will remain in the same order relative to each other.

were three threadgroups which had counts of 4, 1, and 6 respectively for some key, the parallel prefix sum would produce 0, 4, and 5.

Finally, there is the *distribute* phase, which combines parts of the previous two stages. Firstly, it needs to compute the offset for each item within the final, sorted output buffer. It does this by first computing a per-threadgroup offset for each key: the offset for a particular key in the final, sorted buffer will be the total count of all lower-valued keys (which we can compute from the output of the count and scan phases) plus the number of occurrences of that key in all prior threadgroups (given by the scan phase). Once it has a per-threadgroup offset, it can perform a prefix sum within the threadgroup to compute the local offset for each item and therefore the global offset.

Note that in the algorithm described above we need to compute a count for every possible key. In many cases that may not be practical; for example, a 32-bit key necessitates 2^{32} count 'buckets'. This is fairly easily resolved by noting that the sort is required to be stable; therefore, rather than sorting by the entire key, we can instead perform an n-bit sort $\lceil \frac{\log_2(m)}{n} \rceil$ times, where m is the maximum value of the integers being sorted, provided that we sort the least significant bits first. Harada and Howes suggest a four-bit sort in each iteration, necessitating only 16 count 'buckets'; these buckets (each of which is usually a 32-bit integer) can fit in a tractable 64 bytes of GPU memory.

GPU radix sort is a key component of the adaptive and lightmap renderers described in this thesis (Chapter 3) , and is also a basis for the ray direction sorting implementation (Section 5.4). Optimisations to radix sort algorithms from prior work are presented in Section 5.3.

Chapter 3

Architecture of the Lightmap Renderer

At the core of the lightmapper is a Monte Carlo path tracing renderer, responsible for computing the lighting information in the scene and storing it into the lightmap texture.

Rather than directly implementing a lightmap path tracer (for which the output is difficult to validate), the implementation in LlamaEngine (Appendix B.1) was done in stages. First, a camera-based path tracer was built, which was then extended with a renderer that uses adaptive variance-based sampling; that adaptive-sampling framework was then used to build the lightmap renderer. Similarly, this chapter will build upon an understanding of the camera-based and adaptive renderers when describing the lightmap renderer.

In addition to describing the structure of these renderers, this chapter will present a novel method for gather-based filtered sample accumulation and will show how camera-based sampling for lightmap path tracing can be implemented.

3.1 The Path Tracing Framework

Nearly every component in this thesis will be built within the framework of a breadth-first GPU path tracing renderer, loosely based on the implementation in RadeonProRender [43]. Within this chapter, the focus will be predominantly placed on achieving a functional implementation, and will not include modifications necessary for good performance (see Chapters 5 and 6). However, this simple implementation serves as a useful framework for other components, which mostly slightly modify or slot into the structure described.

The following process occurs every iteration (here also referred to as a *frame*). It assumes that the camera position and scene state are fixed.

1. **Generate primary rays:** to begin, a ray is generated for every pixel of the output image corresponding to points on the camera's sensor. For example, when rendering from a camera with perspective projection, the rays have origins at the camera and pass through locations on the camera's near plane.[1]

2. **Compute path radiances using the path tracing estimator:** for each indirect ray:

   (a) **Compute intersections:** find the intersection of the ray with the scene. This intersection could be with scene geometry, or it could be with a skybox or environment map.

   (b) **Add radiance:** add any radiance from the intersection to the *path* corresponding to each ray; for example, emission from an area light.

   (c) **Generate indirect rays:** for the rays that hit scene materials, generate a new ray starting at the surface and with a random output direction. Then, multiply the path throughputs by the surface BRDFs given the input and output directions.

3. **Accumulate samples:** add the total radiance along each path to its corresponding final image pixel in the output accumulation buffer and divide by the total number of frames rendered using Welford's algorithm [57].[2]

Each of these steps is carried out in a separate compute kernel, requiring a separate dispatch from the CPU. Every GPU thread operates on a single path at each step, and no more than one path affects each pixel of the final output image. At no point does the CPU wait on results from the GPU.

Intersection testing is performed using either the RadeonRays [15] or Metal Performance Shaders [58] libraries (Appendix B.2)

3.2 Gather-Based Sample Accumulation

The first extension is to the sample accumulation phase. In the outline given above, one conceptual framing is that every path accumulates its radiance into a single pixel; given the fact that there is exactly

[1] Note that the locations within each pixel should be jittered every frame, since otherwise the output image will be aliased.

[2] Welford's algorithm is a method of computing the running mean and average of a set of samples; in this case, we want the value in the output accumulation buffer to be the running average of all samples taken.

one path for every pixel, this is safe to do. However, it is more useful to frame the process in another way: every pixel *gathers* the sample affecting it at the end of the frame and adds its radiance.

This gather-based model enables some useful alterations. Firstly, image quality can be improved by using filtered accumulation, where each pixel is affected by multiple paths; it is well known that nearest-neighbour interpolation using a box reconstruction filter yields poor visual results (Figure 3.1) [13], and that instead each path should affect a region of pixels. To achieve this, we gather over all paths whose corresponding pixel is in a region surrounding the current pixel, and add each path's contribution with a weight that is a function of its distance from the pixel centre.[3][4] In the model described in Section 3.1, each path is indexed by its nearest neighbour pixel; therefore, gathering over all pixels in an area is fairly straightforward.[5]

For lightmap sampling, filtered accumulation is doubly useful due to the fact that we may not have complete coverage within a texel (i.e., we have 64 fixed sample locations per texel [Chapter 4], whereas geometry may be continuous within a texel). We therefore need some way for samples within a given texel to contribute to neighbouring texels for which all fixed sample locations have missed the geometry. One option would be to use conservative rasterisation, where available, to rasterise the primary ray origins and directions to an intermediate buffer. Another commonly used solution [8] is a dilation filter, which simply fills in each blank texel with data from its neighbouring texel; although this can appear plausible, it is only an approximation. Gather-based accumulation, conversely, is robust from a signal-reconstruction perspective, is not significantly more difficult to implement, and has moderate overhead (Table 3.1).

Additionally, a gather-based model means that we can have multiple samples per pixel or texel per frame. In adaptive sampling (Section 3.3), for example, there may be multiple samples concentrated on the pixels with the highest variance, or in lightmap path tracing samples may be focused on areas of the lightmap which are visible within the user's current view. As such, a model that uses scattered writes with multiple samples per write location necessitates atomic operations. Floating point atomic writes are not generally supported in hardware and must be emulated with integer compare-and-exchange. In addition, we usually want to accumulate multiple pieces of data per pixel or texel; computing the vari-

[3] Paths are generated at offsets within pixels for antialiasing, and that offset needs to be accounted for in the filtered accumulation.

[4] In LlamaEngine (Appendix B.1), the supported filter types are Box, Gaussian, Mitchell, and Lanczos-Sinc filters with a 1.5 pixel support radius (based on the implementation in PBRT [13]), although theoretically any filter with any support radius is supported.

[5] Since there is in this case a one-to-one mapping of pixels to paths, gathering over an area is a matter of selecting the pixels to gather and using the mapping to find the corresponding path for each.

Figure 3.1: Filtered sample accumulation in LlamaEngine (Appendix B.1). Scene is 'The Wooden Staircase' by 'Wig42' [59].

(a) Box filter

(b) Gaussian filter with a two-pixel radius

(c) Mitchell filter with a two-pixel radius

The box filter provides the worst reconstruction, with harsh, aliased edges. The Gaussian filter provides a smooth image at the cost of lost texture detail, while the Mitchell filter strikes a reasonable balance, retaining detail while mitigating aliasing.

Table 3.1: Timing per frame for filtered vs. non-filtered accumulation at 2560×1440 resolution.

Filtered accumulation	$1.39ms$
Non-filtered accumulation (box filter)	$0.86ms$
Filtering overhead	$0.53ms$

Filtered accumulation uses a Gaussian filter with a 3x3 pixel footprint. Adaptive rendering is disabled, so there is one path per pixel.

ance using Welford's algorithm [57], for example, requires exclusive access to both the running mean and variance, effectively requiring a lock in the presence of thread contention. A similar situation exists with accumulation for spherical Gaussians, Ambient Dice, and other non-orthogonal basis functions (Chapter 7).

In prior work, sample accumulation has been performed in a number of ways. RadeonProRender uses scattered atomic writes for adaptive rendering; similarly, the OpenCL path tracer LuxRender [60] uses atomic writes to accumulate filtered results. PBRT accumulates results into tiles, with one thread operating on each tile (and therefore on all pixels within that tile); a global mutex coordinates filtered accumulation to the final output image. All of these approaches have some issues in the context of lightmap path tracing. As already mentioned, atomic operations conflict with algorithms that compute the variance or update the mean in place (rather than accumulating all the results and then dividing by the sample count at the end). On the other hand, PBRT's approach of per-thread tiles doesn't scale to the large number of threads inherent in a vectorised or GPU-based implementation. A gather-based model resolves these issues.

3.2.1 Offset Buffer Generation

Removing the restriction that there is one path per pixel presents other issues, however; namely that finding all of the paths affecting a given pixel becomes substantially trickier. In this section, I propose an efficient method to generate an *offset buffer*. In the context of sample accumulation, the offset buffer gives the index of the first path and the total number of paths affecting a given pixel, enabling a gather operation to be performed over all of those paths.

As a prerequisite to generating the offset buffer, the paths must be in sorted order relative to the indexing scheme used for the image (e.g. linear row-based indexing or tile-based indexing [Section 5.2]). This can be accomplished by first generating a buffer of indices for which to trace paths, sorting that buffer

using e.g. GPU radix sort (Section 2.5), and then generating paths according to the indices in that buffer, where each index is the discretised nearest-neighbour of each path.

Formally, the offset buffer is an integer array such that *offsetBuffer*$[i]$ contains the index of the first element x within some source buffer, where there exists some mapping $f(x)$ such that $f(x) = i$. For the offset buffer to exist, the source buffer must be sorted by $f(x)$.

If there is no element in the source buffer such that $f(x) = i$ for a given i then *offsetBuffer*$[i]$ should contain the offset for the next highest integer j for which $f^{-1}(j)$ exists in the source buffer. This property means that the number of elements for which $f(x) = n$ given some n is equal to *offsetBuffer*$[n + 1] -$ *offsetBuffer*$[n]$.

The offset buffer can also be viewed as the exclusive prefix sum of the histogram of the mapped values i. In fact, one method to produce an offset buffer would involve computing a histogram in parallel on the GPU and then performing a parallel prefix sum over that histogram.

Computing a histogram and then prefix sum of a large buffer is expensive; using AMD's *Parallel Primitives* library [15] for a buffer containing 262,144 elements requires five dispatch calls to the GPU for the prefix sum alone. However, by modifying the definition of the offset buffer slightly and adjusting the code that reads from it accordingly we can reduce the offset buffer computation to an inexpensive two-pass process comprised of a fill using the GPU's blit/copy queue and then a simple compute kernel. The computation scales linearly with the maximum possible value of $f(x)$ for the elements in the source buffer (which is usually the number of texels within the accumulation buffer).

The key observation is that, although $f^{-1}(i)$ is a many-to-one mapping (for example, many paths may affect the same pixel), there is no more than one j such that $f(x_j) > f(x_{j-1})$ for each possible output of $f(x)$ (in other words, there is at most one first and one last path affecting each pixel). We can therefore perform a single pass over the source buffer, writing j to the offset buffer at $f(x_j)$ whenever $f(x_j) > f(x_{j-1})$.

Given this description, there remains the issue that not every possible value within range of $f(x)$ is guaranteed to be in the source buffer; there may be gaps. If there is no element for which $f(x) = j$ in the source buffer, *offsetBuffer*$[j]$ must contain the offset for the next highest integer k for which $f^{-1}(k)$ does exist in the source buffer; however, to do this in a compute shader requires a single thread to scan either backwards or forwards over multiple elements and write the offsets into the offset buffer, yielding quadratic worst-case performance.

We can resolve this with two changes. Firstly, each thread j for which $f(x_j) > f(x_{j-1})$ should write the offset j to both $f(x_j)$ and $f(x_{j-1}) + 1$; often the two write locations will be the same, but if there are gaps in the source buffer both writes are necessary. Secondly, we fill the offset buffer with some sentinel value (e.g. `0xFFFFFFFF` for 32-bit integers) before running the compute shader so that each element of the offset buffer that is not written to by the compute shader holds the sentinel value.

When reading, we can use that sentinel value to identify missing entries in the offset buffer. If any of the following are true, the number of elements in the source buffer for which $f(x) = n$ given some n is 0:

- *offsetBuffer*[n] is equal to the sentinel value.
- *offsetBuffer*[n + 1] is equal to the sentinel value.
- *offsetBuffer*[n + 1] is equal to *offsetBuffer*[n].

Otherwise, all elements in the source buffer in the range [*offsetBuffer*[i], *offsetBuffer*[i + 1]) have a value $f(x)$ such that $f(x) = i$, as was desired.

This generation method (shown in full in Listing 3.1) is highly efficient. Generating the offset buffer for a 2560 $\times$ 1440 output image (with 3,686,400 pixels) on the test hardware (Section 1.3) takes 145μs for the compute shader; timings for filling the buffer with the sentinel values are not available due to its being executed on the GPU's copy queue.

Listing 3.1: Offset Buffer Generation

```
// Input is a sorted buffer of texel indices belonging to particular paths.
// One thread should be dispatched per path, rather than per texel index.
kernel void generateOffsetBuffer(
                                const uint texelIndexCount,
                                const uint *pathTexelIndices, // the texel indices for each
                                    path.
                                uint *offsetBuffer, // must be initialised to some flag
                                    value (e.g. 0xFFFFFFFF) before this.
                                uint *pathCountOut, // containing the active path count
                                uint threadId [[ thread_position_in_grid ]],
                                uint pathCount [[ threads_per_grid ]]
                        ) {

   if (threadId >= pathCount) {
      return;
   }
```

```
    // Thread i looks at paths i - 1 and i
    uint pathIndex = threadId + 1;

    // Fetch the texel for the previous path...
    uint previousPathTexel = pathPixelIndices[pathIndex - 1];
    // and for the current path
    uint currentPathTexel = pathIndex == pathCount ? tileIndexCount :
        pathTexelIndices[pathIndex];

    // If this path is different from the previous path and the previous addresses a valid
        texel...
    if (previousPathTexel < currentPathTexel && previousPathTexel + 1 < texelIndexCount) {
        // fill the offset buffer for the index after the previous path
       offsetBuffer[previousPathPixel + 1] = pathIndex;

       // and the index for the current path (which may be the same as previousPathPixel + 1)
       if (currentPathPixel < texelIndexCount) {
          offsetBuffer[currentPathPixel] = pathIndex;
       }
    }

    // If the thread id is 0 (i.e. it's the first thread, initialise the first element of the
        offset buffer to 0.
    // Additionally, fill the index in the offset buffer corresponding to this thread's
        element, since no earlier thread can have done it for us.
    if (threadId == 0) {
       uint currentPathPixel = pathPixelIndices[0];

       if (currentPathPixel < texelIndexCount) {
          offsetBuffer[currentPathPixel] = 0;
       }
    }

    // We want to know how many active paths there are.
    // The following condition will only be true for one thread.
    if (currentPathTexel >= texelIndexCount && previousPathTexel < texelIndexCount) {
       uint activeCount = pathIndex; // The last path was pathIndex - 1, so pathIndex is the
           number of active paths.
       *pathCountOut = activeCount;
    } else if ((threadId == 0 && previousPathTexel >= texelIndexCount) {
       // There were no active paths.
       *pathCountOut = 0;
    }
}
```

3.2.2 Gather-Based Accumulation with Variable Paths per Pixel

At this point, we can bring together the offset buffer generation algorithm and the gather-based method to implement a method for filtered accumulation of sample values in parallel on the GPU.

The process for gather-based sample accumulation (including its dependencies) is as follows, with every step being carried out on the GPU using compute shaders. The outline follows that given in Section 3.1, with new stages *italicised*.

1. ***Select a (continuous) output location for each path*** *within a given frame; for example, path 0 could be at location (0.5, 0.5) within texel (0, 0).*

2. ***Sort the paths*** *(e.g. using a GPU radix sort [Section 2.5]) by their discretised (nearest-neighbour) output texels.*

3. ***Generate an offset buffer*** *such that* offsetBuffer*[*i*] returns the index of the first path within the paths buffer whose nearest neighbour is texel* i*, or a sentinel value if there is no such texel. The offset buffer must have capacity for the maximum possible texel index.*

   (a) *Firstly,* ***clear the offset buffer*** *to a sentinel value using the GPU's blit queue. The sentinel value must be outside the range of valid texel indices; for example, for 32-bit indices the maximum unsigned integer* `0xFFFFFFFF` *can be used.*

   (b) *Then,* ***perform the offset buffer generation algorithm*** *(Listing 3.1), dispatching at least one thread per path.*

4. **Generate primary rays.**

5. **Compute path radiances using the path tracing estimator.**

6. **Accumulate samples** *(modified)* (Listing 3.2): for each texel in the output image, gather radiance *from affecting paths* and write to the accumulation buffer.

   - At the start of the compute shader, the current total mean *and weight* for the current texel is either read from the accumulation buffer or initialised to zero.
   - *When using a single-texel box filter, the affecting paths are all those paths whose nearest neighbour are the current texel.*

- *If accumulating within a radius, all paths whose nearest neighbours are texels within the filter radius of the current texel must be considered, with their contributions weighted according to the filter kernel.*
- Once the mean *and total weight* have been updated, each thread writes back the new values for its texel to the accumulation buffer.

Listing 3.2: Accumulation by Offset Buffer Sampling

```
float4 meanAndWeight = ...; // The mean and weight for the current texel.
unsigned *texelIndicesToPathIndices = offsetBuffer;

for sourcePixel in neighbourhood(currentPixel) {

  unsigned texelIndex = pixelCoordToTexelIndex(sourcePixel);
  unsigned basePathOffset = texelIndicesToPathIndices[texelIndex];

  if (basePathOffset == kSentinelValue) {
    return; // No valid paths for the current pixel.
  }

 // The index for paths at the pixel after the current one. The delta gives the count.
 uint nextPixelPathOffset = (texelIndex + 1 >= texelIndexCount) ?
     texelIndexCount : texelIndicesToPathIndices[tileIndex + 1];

 if (nextPixelPathOffset == kSentinelValue) {
    return; // No valid paths for the current pixel.
 }

 for (uint pathIndex = basePathOffset; pathIndex < nextPixelPathOffset; pathIndex += 1) {
    float2 pathSampleLocation = pathSampleLocations[pathIndex];
    float sampleWeight = filterWeight( pathSampleLocation - currentPixelCentre);
    AddPathSample(paths[pathIndex], sampleWeight, &meanAndWeight);
 }
}
```

3.3 Adaptive Rendering

Monte Carlo integration is a progressive process wherein samples are continuously taken until the result converges. However, not every pixel will converge at the same rate. In camera-based rendering, for example, if a primary ray points at the skybox, the variance between multiple samples pointing at the

skybox is likely to be extremely low. If, however, paths along that ray often take routes with low probability but high radiance, those paths will have high variance, and will manifest as noisy parts of the image.

Ideally, we want to produce a noise-free image as quickly as possible. Given that goal, it does not make sense to equally spread effort across all texels. Instead, we can adaptively sample different texels based on some metric such as their variance (Figure 3.2).

An initial framework for this was provided by Lee et al. (1985) [61], who derived a relationship between the number of sample rays and the quality of the estimate and concluded that the number of sample rays needed is dependent upon the variance of the estimate. Hillaire (2018) [4] describes a similar approach for adaptive sampling wherein the variance of each texel is computed during sample accumulation and then that per-texel variance is used to distribute new samples.

As per Hillaire, we can accumulate the running variance into a separate buffer during sample accumulation using Welford's algorithm [57]. Then, at regular intervals (e.g. every 64 frames), the variance within each 8×8 tile is averaged and output to a new buffer.[6] A prefix sum over that buffer is then performed to generate a probability distribution for sampling. In my implementation, the buffer is asynchronously read back to the CPU, and the probability distribution is generated there. However, it is also fully possible to generate the distribution on the GPU: performing a parallel prefix sum over the elements in the buffer will generate a cumulative probability distribution that can then be sampled from using a binary search.[7]

This adaptive sampling takes place in a new pass at the start of the path-tracing process. Within the pass, each thread probabilistically selects a pixel index and then outputs that to a buffer.

When rendering adaptively, the number of paths per texel is variable: in a given frame, multiple paths may affect a single texel, while some texels may have no paths traced. In addition, the paths will not be generated in sorted order. These factors necessitate a number of alterations to the sample accumulation pipeline, which are discussed in depth in Section 3.2.

There are some important caveats with adaptive sampling. Kirk and Arvo [62] showed that adaptive sampling algorithms that both accumulate a sample set and use that sample set to estimate how many more samples need to be taken are biased and do not converge to the true result; to mitigate this, the variance should be calculated for a set of samples whose contributions are later discarded. In many cases, the slight bias introduced is worthwhile given the lesser computational expense.

[6] The tile size was chosen to be 8×8 since there are 64 threads in a SIMD group on the AMD test hardware, and thus computing the average for a tile using SIMD operations is straightforward and does not require threadgroup operations.

[7] Note that, in this scenario, each uniform random sample must be multiplied by the total before performing the binary search.

Similarly, Mitchell [63] noted filtered accumulation (Section 3.2) will have incorrect results in the presence of adaptive sampling. Consider the case where a particular texel is given very few samples while its contributing neighbours are given many: the end result for that texel will have insufficient contributions from within its own region.

3.4 Lightmap Path Tracing

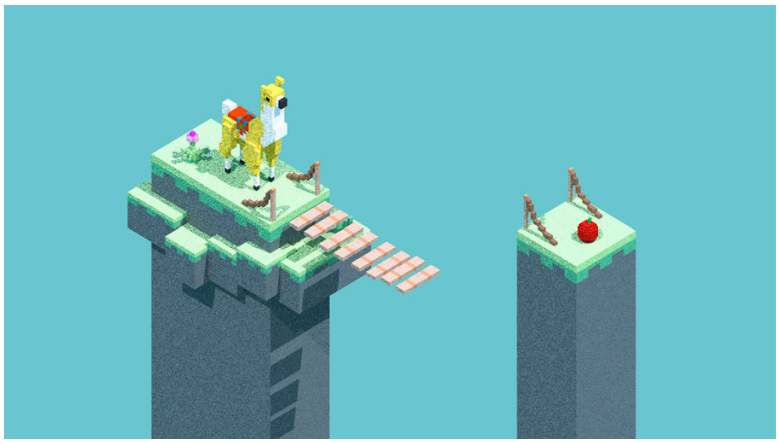
Since we cannot sample the entire lightmap every frame and maintain interactive framerates, only a subset of the lightmap can be rendered each frame. To provide the best user experience, texels are randomly sampled on the lightmap (as opposed to, for example, rendering the top-left of the lightmap in one frame and the top-right in the next), ensuring that a preview for the entire lightmap (and therefore scene) is available as quickly as possible.

Given this adaptive approach, the pipeline for lightmap path tracing deviates only slightly from that of adaptive rendering (Section 3.3). As with adaptive rendering, a probability distribution is used to determine the pixel or texel each path operates on (or, for filtered accumulation, the nearest neighbour of the path). The initial probability distribution should have equal probability for all texels with at least one valid sample location and zero probability for all others; sampling according to the number of valid samples per texel results in undersampling for texels with few valid sample locations. This probability distribution may be multiplied by the per-texel or per-tile variance as samples are accumulated to perform adaptive lightmap path tracing.

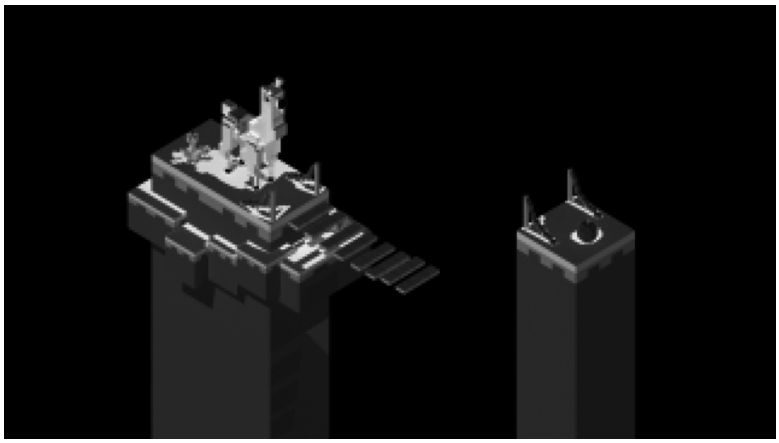
When generating primary rays, an instance-index and triangle-index pair are chosen per-texel. That triangle's data is then decoded from the mesh's vertex buffers, giving a world-space position, world-space normal, and lightmap UV for each of its three vertices. The barycentric coordinates within the triangle are determined by the location within the texel (and therefore the lightmap) and the triangle's lightmap UVs; high-precision calculations should be used in calculating the barycentric coordinates to avoid invalid values. Given the barycentric coordinates, a world-space position and normal can be interpolated from the per-vertex values (Section 4.3).

The primary ray's position is given by the interpolated world-space position. If scalar accumulation is being used (accumulated cosine-weighted irradiance), then the ray direction is given by cosine-weighted sampling of the hemisphere around the surface normal and the sample values may be directly accumu-

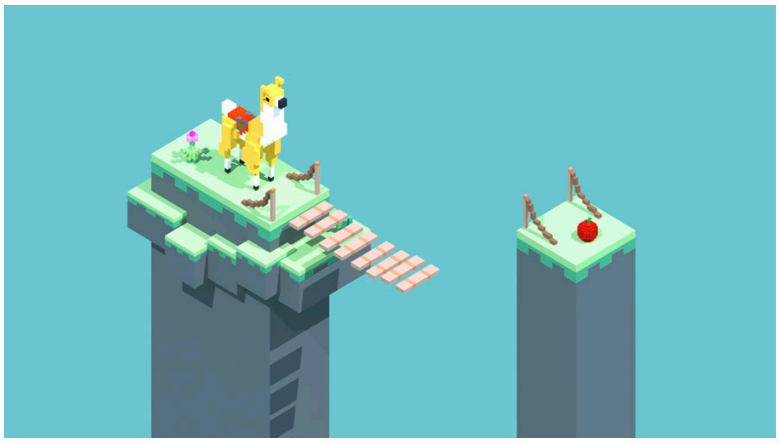
Figure 3.2: Variance-Based Adaptive Sampling



(a) The path-traced image with one sample per pixel



(b) Samples taken per pixel of the converged image, where lighter means more samples taken.



(c) The final converged image.

Note how many samples are taken in the noisy areas of the image with high indirect contribution, while fewer are taken in the low-variance, directly lit areas.

lated.[8]

Regardless of whether only diffuse irradiance is being accumulated or whether an indirect specular representation is as well (Chapter 7), all materials and all material layers should always be sampled in the path tracing process for an accurate result, not just the diffuse layer. Specular to diffuse transport[9] can be an important visual component of some scenes and requires no additional effort to encode once the path tracer has support for specular materials.

Accumulation into Linear Bases

If, instead of accumulating scalar cosine irradiance, the goal is to encode into a linear basis $\sum_i b_i B_i(s)$ so that radiance or irradiance may be reconstructed in any direction (Chapter 7), two changes must be made. The first is that the rays should be generated uniformly on the hemisphere rather than in a cosine-weighted fashion; the second is that the basis coefficients b_i should be stored into separate textures. Note that accumulation with Welford's algorithm requires keeping track of the accumulated weight per-pixel; this can be done by either storing the weight in the alpha channel of one of the accumulation textures or by using a separate 32-bit floating point buffer.[10]

3.4.1 Interactive Lightmap Path Tracing

A major draw of GPU-based lightmap path tracing is that the user can continue to interact with and navigate the scene while the path tracing process is taking place. In a single-GPU environment, rendering the main view competes for resources with the path tracer, necessitating some sort of load balancing to ensure that the main view continues to render at interactive framerates.

This is relatively straightforward within the adaptive rendering framework. In adaptive rendering, you provide a maximum number of paths to trace within each frame, and the texels to operate on are probabilistically chosen. The problem of maintaining an interactive framerate can therefore be partially addressed by continuously adjusting the maximum path count; however, due to the resource contention and variable rendering workload as the view changes it is inevitable that there will be some frame cadence

[8] This is because the PDF (probability density function) for a cosine-weighted ray is $\frac{\cos(\theta)}{\pi}$, which exactly matches the Lambertian diffuse BRDF; dividing by the PDF cancels with multiplying by the BRDF.

[9] An example of specular to diffuse transport is light bouncing off a mirror to illuminate the diffuse ground; another example is specular transmission through a glass window hitting a diffuse surface.

[10] Real-valued weights exist due to filtered accumulation and 16-bit floating point is insufficiently precise.

issues. Apers et al. note [1] that this can be mitigated in a dual-GPU setup where one GPU is exclusively dedicated to lightmap path tracing; however, such a setup was not available for me to test.

In my implementation, load balancing is done using a multiple-frame response, where the path count for the next frame is determined by the exponential moving average frame completion time t:

$$t^{(i+1)} = 0.95t^{(i)} + 0.05 \times \textit{frameTime}(i) \tag{3.1}$$

The path count p is then given by:

$$p^{(i+1)} = p^{(i)} \cdot \sqrt[3]{\frac{\textit{targetFrameTime} - \epsilon}{t^{(i+1)}}} \tag{3.2}$$

ϵ is some bias that aims to make sure we exceed the target framerate rather than hovering above or below it. As an example, if a 30FPS target is desired, the target frame time including ϵ might be around $\frac{1000}{33}ms$ rather than $\frac{1000}{30}ms$.

In practice, this simple measure does fairly well at maintaining the target framerate. In some cases it is also worth introducing a maximum and minimum number of paths; a maximum ensures that the response doesn't cause the GPU to be momentarily overloaded with work, while a minimum ensures that a useful amount of work is being submitted every frame.

3.4.2 Camera-Based Lightmap Sampling

Rather than randomly sampling all valid lightmap texels, Hillaire suggests [4] focusing on the areas currently visible to the user's camera (Figure 3.3), although an implementation design is not provided. When interactively navigating the scene, this can have a dramatic positive effect on the rate of convergence from the user's perspective (Figure 3.4); in addition, it prevents samples from being taken in areas with valid geometry that is permanently occluded from the user's view. Camera-based sampling also helps to ensure that path origins are reasonably coherent in world space, improving ray-tracing performance for the primary rays.

One of the earliest steps in the adaptive path tracing process is to generate the *texel domain buffer*: a list of all texel indices from which paths will be traced in the current batch. Previously, this was done by randomly sampling from a probability distribution, where each texel either had a binary probability (i.e. either it does or does not have valid samples) or a probability based on its variance. However, the

list of texels to trace paths from can also be generated through either a rasterisation or ray tracing process. In my implementation, the scene is rasterised from the user's view to a texture whose pixel count is approximately the path count:

$$width = \lfloor \sqrt{pathCount \times aspectRatio} \rfloor$$
$$height = \lfloor \sqrt{\frac{pathCount}{aspectRatio}} \rfloor$$

The render texture has a single-channel 32-bit `uint` pixel format. In the fragment shader, the pixel coordinate is output as a texel index (i.e. a single number generated either by linear row-based or by tile-based [Section 5.2] indexing) representing the lightmap texel, where the lightmap texel is determined from the lightmap dimensions and the $[0, 1)$ lightmap UV passed through from the vertex shader. Once rasterised, this texture is copied to a buffer, which is then passed through to the remaining stages.

Note that, due to floating-point imprecision, not every lightmap texel within view may actually have valid samples in the buffer generated in Section 4.2. In the fragment shader, therefore, the texel index is checked against the sample buffer to ensure that the ray generation stage will successfully find geometry in that texel; more specifically, the number of bits set in the valid samples mask for the texel must not be zero. If it is, `0xFFFFFFFF` is output as the texel index. Similarly, the output texture is cleared to `0xFFFFFFFF` before rasterising the geometry. The radix sort phase sorts the invalid samples to the end of the sorted path indices, which the offset buffer generation process (Section 3.1) then discards.

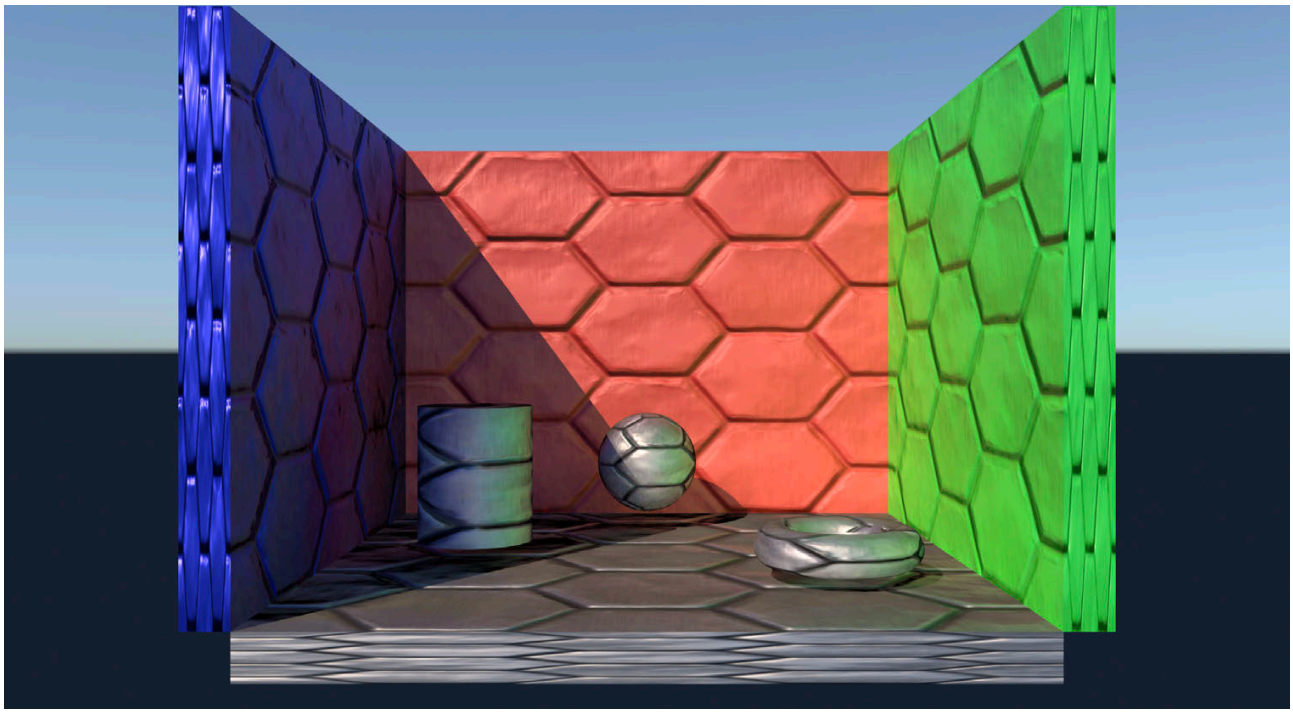
If the render texture resolution is coarser than the lightmap resolution (i.e. if there are multiple lightmap texels per render texture texel), it is possible that some lightmap texels will be missed given a fixed camera position. To combat this, the projection matrix should have a subpixel jitter applied every batch in the same way as is done for temporal antialiasing (Karis 2014) [64].

If there are translucent lightmapped surfaces in the scene (e.g. glass with lightmapped specular) this approach presents a challenge, since both the translucent objects and any objects behind it need to have lightmap samples generated. As a simple workaround, translucent objects can be stochastically hidden in this view (e.g. with 0.5 probability of being rasterised each frame), ensuring that both the translucent objects and any objects behind them both receive lightmap samples.

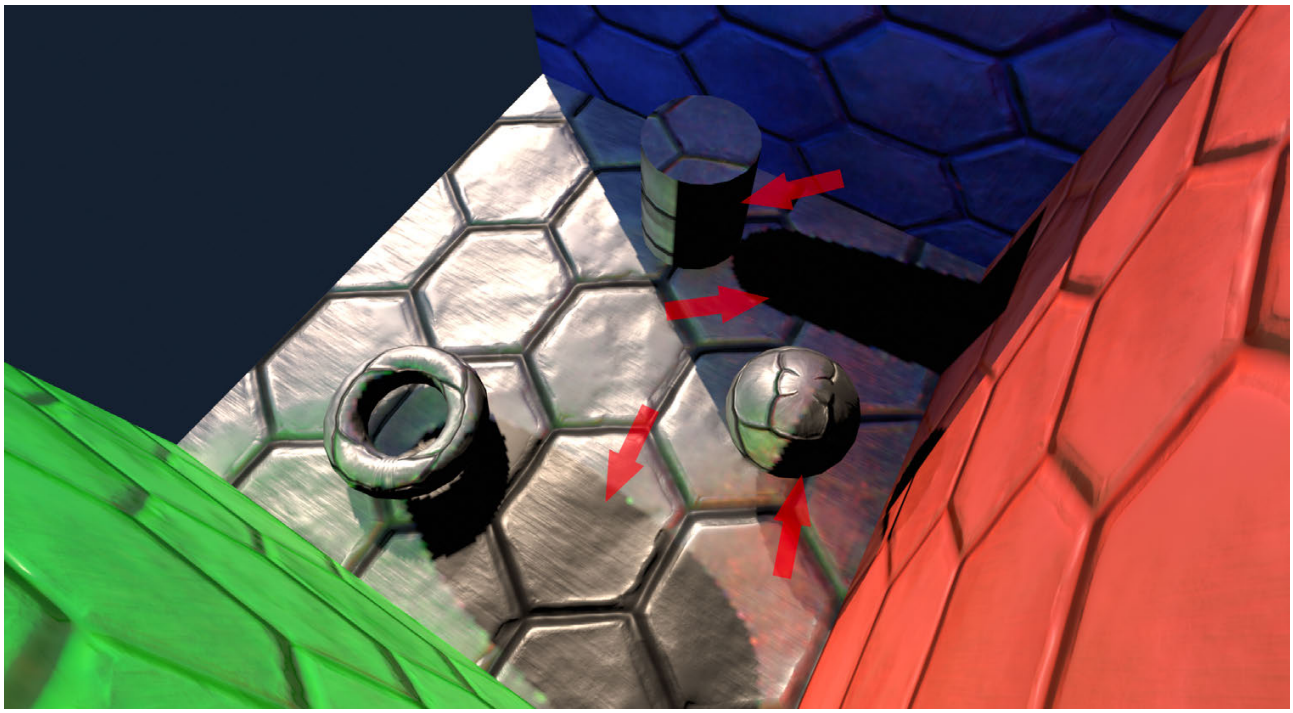
As an extension to camera-based sampling, it is possible to perform camera-based variance-based sam-

pling (Section 3.3) for lightmaps. In addition to outputting the texel indices, the variance for each rasterised pixel should be simultaneously written to a separate render target, which is then used to generate a per-texel probability distribution as described in Section 3.3. That probability distribution replaces the per-texel probability distribution used when performing non-camera-based sampling; when a pixel is chosen, its index is read from the render target containing the texel indices corresponding to the distribution, and that texel index is then output to the texel domain buffer. In my implementation, non-adaptive camera-based sampling performs very well, obviating the need to also perform variance-based adaptive sampling (which carries a higher computational overhead and implementation complexity).

Figure 3.3: A demonstration of camera-based sampling for lightmap baking. Scene credit The Baking Lab [18].

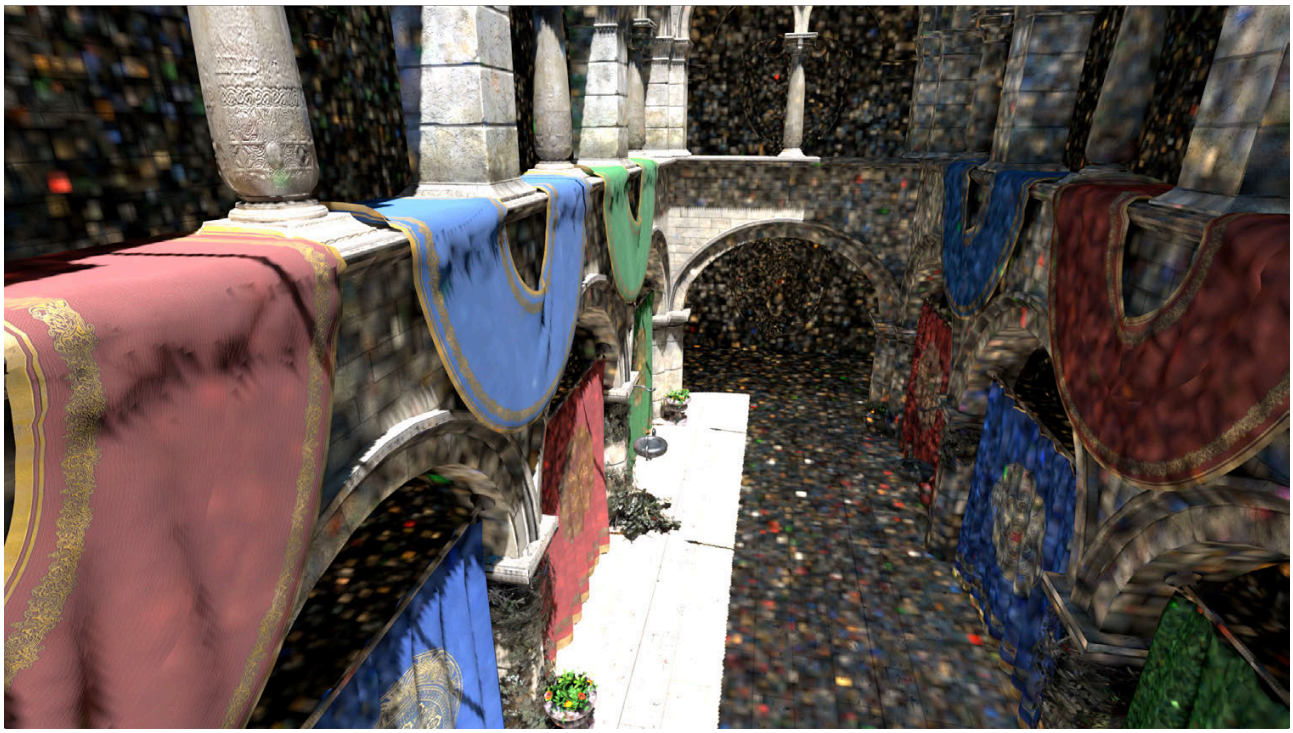


(a) Sampling is performed from the camera's perspective, yielding a rapidly converging image without visual artefacts.



(b) Baking is paused and the camera is moved. The non-sampled parts of the lightmap invisible from the previous camera's perspective (indicated with red arrows) contain no indirect radiance information.

Figure 3.4: Convergence rate of camera-based vs. non-camera-based lightmap sampling.



(a) Lightmap texels are uniformly randomly chosen to be sampled.

(b) Sampling is performed from the camera's perspective, yielding a rapidly converging image.

Both images use 921,600 samples per frame across 64 frames and the PMJ-02 sampler.

Chapter 4

Lightmap Parameterisation

To store data such as indirect illumination in a lightmap (Section 2.1), we need a *parameterisation* for every object in a scene, mapping from points on meshes to locations within the lightmap texture. This chapter will detail the parameterisation process developed for use in LlamaEngine (Appendix B.1), expanding on the method used in EA's Frostbite engine [8] with implementation details and performance considerations. It will also detail how that parameterisation can be used on the GPU to generate primary rays for path tracing the lightmap.

In this chapter, a *mesh* will refer to a set of vertices, each carrying some possibly-unique information, which are indexed into triangles. An *object* will refer to a set of one or more instanced meshes, with a unique transform applied to the meshes' vertex positions. A single mesh may appear multiple times in a scene as part of different objects; each object appears only once.

4.1 Packing the Lightmaps

Since we assume that all meshes in a scene are comprised of triangles, parameterising the meshes into a lightmap amounts to finding a unique *lightmap UV* for every vertex, identifying the location within the lightmap that the vertex's data is stored. That lightmap UV, interpolated across the triangle to the shading point, is used to fetch the data from the lightmap texture.

There are two main approaches to parameterising our scene. The first is to treat it as a global problem: consider every mesh of every object independently and generate a parameterisation for the entire scene using only a set of world-space vertices and the connectedness between those vertices. This approach is

Figure 4.1: The parameterised lightmap for Sponza Atrium [5]

Each colour represents a separate mesh; however, each mesh may be composed of multiple charts (planar unwrappings of sections of the mesh) which are distributed throughout the lightmap.

both computationally expensive and limits the ability to cache data, meaning that scaling a single object or adding a new object to the scene necessitates restarting the entire process; it also means that every mesh needs a unique per-vertex stream of lightmap UVs, thus complicating instanced drawing.[1]

Rather than performing a global solve, I follow the basic method outlined by O'Donnell in *Precomputed Global Illumination in Frostbite* (2018) [8]. O'Donnell suggests first parameterising each mesh into a series of *charts*, where each chart is a continuous unwrapping of some subset of the mesh's vertices. Each vertex then stores its position within its lightmap chart (in the range [0, 1] in X and Y) along with its chart index. Given a set of charts for each mesh, every object in the scene can be packed into the lightmap according to the world-space scale of that object, with the parameterisation being expressed by a per-object per-mesh-chart scale and offset to the lightmap UVs. Using this approach, the chart parameterisations can be cached per mesh, and only the packing of the charts into the lightmap needs to be computed on a per-scene basis.

Generating high-quality two-dimensional planar mappings for 3D meshes is a very difficult problem and not one I sought to solve for my thesis. To generate the charts, I made use of the open-source tool *Thekla Atlas* (Castaño 2013) [65] with some slight modifications (namely, to avoid placing margins on the edges of charts, to output the mesh-space per-chart sizes, and to avoid packing the charts for each mesh into a single, larger atlas). I integrated Thekla Atlas into the content pipeline for LlamaEngine, enabling near-automatic mesh parameterisation.

Given a parameterisation of each mesh into charts, O'Donnell's process of generating a parameterisation for the full lightmap is as follows:

1. For every instance of every mesh in the scene, compute the size in texels of its charts given a target texel density per world space unit; assuming the chart is area-preserving, this will ensure that triangles are allotted lightmap area proportional to their area, resulting in relatively consistent density across the entire lightmap.

2. Inset the edges of each chart by half a texel to avoid light bleeding due to bilinear filtering. At this step, every chart for every instance has a scale for its lightmap UVs.

3. Conservatively rasterise every triangle in every instance into bitmap images for each chart, resulting

[1] Instanced rendering is where a mesh is reused in multiple locations in the scene, exploiting the fact that the data can be split into per-vertex and per-object information. For example, a mesh duplicated into two different places in a scene can be expressed using the per-mesh vertex positions and a per-object mesh-to-world transform.

in one bitmap per chart per instance. This step can be done in parallel for all instances.

4. Sort all charts by their perimeter from largest to smallest.

5. Classify charts into size buckets based on their perimeter; for example, I define the bucket to be $\lfloor \frac{1}{16} \rfloor^{th}$ of the perimeter.

6. Iterate through every chart and try to place it within a bitmap image (Section 4.1.1) of the lightmap.

    - This amounts to finding the first origin pixel where placing the chart at that pixel would not result in an intersection with charts already in the lightmap.
    - If the chart is in a smaller bucket than the previous chart, begin searching from the first row of the image; otherwise, begin searching from immediately after where the previous chart was placed. Doing this allows small charts to fill the gaps left between larger charts without requiring a brute-force search for every chart.
    - If the chart cannot be placed, resize the lightmap and try again.

The end result of this process is a scale and offset for the lightmap UVs within each mesh for each instance. This scale and offset should map all UVs into the range $(0, 1)$ for each axis, where the input (unscaled) UVs are in the range $[0, 1]$. Figure 4.1 shows an example of the resulting parameterisation when applied to the lightmap UVs for each instance.

4.1.1 BitImage

The charts for each mesh instance are packed into a bitmap image, where the image stores whether each texel is currently occupied. The design of the bitmap image data structure is not specified in prior work and has performance implications for the lightmap parameterisation process.

The most trivial implementation of a bitmap image is to use one byte to store every bit, and to store those bytes in a linear array. However, this design is fairly slow when used to find an insertion position within another image, such as is done when placing charts within the lightmap. Instead, we can achieve a speedup of around 60% by instead storing the data within individual bits of machine-width integers and using bitwise operations (Table 4.1).

Within this `BitImage`, every 2D pixel coordinate maps to an integer index and a bit offset within that integer:

$$\text{bitIndex}(x, y) = y \times \textbf{storageWidth} + x \tag{4.1}$$

$$\begin{aligned}(\text{uintIndex, bitOffset}) = (&\lfloor \frac{\text{bitIndex}(x, y)}{\textbf{bitWidth}} \rfloor, \\ &\text{bitIndex}(x, y) \ \textbf{\textit{mod}} \ \textbf{bitWidth})\end{aligned}$$

storageWidth is given by rounding up the width of the image to the nearest multiple of the integer's bit width (e.g. 64 bits on a 64-bit architecture). This ensures that the start of every row is aligned to an integer word, making insertion testing more efficient.

4.1.2 Rasterising the Charts

The output of Thekla Atlas is a per-vertex chart index and UV within that chart for every mesh. Prior to inserting these charts into our lightmap, each chart must be rasterised into a `BitImage`; this is done by iterating over every triangle in the mesh and assigning it in the chart image.

Complicating matters is the fact that lightmaps are generally sampled using linear interpolation, meaning lighting information from one texel can 'bleed through' to neighbouring texels. To avoid this, when rasterising triangles to a chart, a one-pixel margin is generally [8] left around each texel to prevent different geometry from sampling each other's lighting information.

To assign a triangle to a chart, we first compute a bounding box around the pixels that triangle affects, expanding the bounding box by one pixel on all sides to allow for the margin. Then, for every pixel within that bounding box, a box is formed around that pixel; if the pixel's centre is at $(x + 0.5, y + 0.5)$, the box's minimum point is at $x - 0.5 + \epsilon, x + 1.5 - \epsilon$, where ϵ is a small tolerance to prevent a UV exactly at a pixel's centre from affecting its neighbours. A triangle-box intersection text is then performed; if the triangle overlaps the box, that pixel is marked as filled in the `BitImage`.

4.1.3 Inserting the Charts

Testing whether a chart can be inserted into the lightmap at a particular position can be done using bitwise operations. A chart can be inserted if the bitwise AND of every pixel in the chart with the pixel at

Table 4.1: Timing for packing a 2048×1450 lightmap for Sponza Atrium.

ByteImage (one byte per pixel)	$2910ms$
BitImage (unaligned, without padding)	$1780ms$
BitImage (aligned, padding at the end of each row)	$1090ms$

its insertion location in the lightmap is equal to zero; that is, if there are no pixels that are set both in the lightmap and in the chart.

If the `BitImage` format is tightly packed, with no padding on the end of rows, testing insertion for rows other than the first may involve performing an unaligned load from two integers in the backing storage since the first pixel in a row may not be the first bit in an integer. If, however, we pad the end of each row with bits set to zero, every row in the charts will be aligned to integer boundaries, improving testing performance at a very slight increase in memory usage.

Regardless of whether the charts are padded, we still need to be able to test at arbitrary insertion positions for the lightmap, which means performing unaligned loads for the lightmap's pixels. To do this, we can bitshift and bitwise OR together the contents of adjacent memory locations. The `BitImage`'s storage is given one extra zeroed word at its end to enable us to read at an offset within the last index without bounds checks or risking buffer overruns.

Listing 4.1: BitImage Insertion Testing and Insertion

```
extension BitImage {
  public func canInsertDisjoint(image: BitImage, x: Int, y: Int) -> Bool {
     if image.width > (self.width - x) || image.height > (self.height - y) {
        return false
     }

      // Compute how many words we need to stride through for each row.
      // NOTE: we use 'self.width' rather than 'self.storageWidth' since we don't care about
          the zero bits in the padding at the end of the row.
      let elementsPerRow = (image.width + UInt.bitWidth - 1) / UInt.bitWidth

     for row in 0..<image.height {
        let selfBaseOffset = self.bitIndex(x: x, y: y + row)
        let (selfIndex, selfOffset) = selfBaseOffset.quotientAndRemainder(dividingBy:
            BitImage.bitsPerElement)

        let otherBaseOffset = image.bitIndex(x: 0, y: row)
        let otherIndex = otherBaseOffset / UInt.bitWidth

        let secondWordShift = UInt.bitWidth - selfOffset
```

```
        for column in 0..<elementsPerRow {
            let firstWord = self.storage[selfIndex + column]
            let secondWord = self.storage[selfIndex + column + 1]

            let selfTest = (firstWord >> selfOffset) | (secondWord << secondWordShift)
            let otherTest = image.storage[otherIndex + column]

            if selfTest & otherTest != 0 {
                return false
            }
        }
    }
    return true
}

// Precondition: canInsertDisjoint(image, x, y)
public mutating func insert(image: BitImage, x: Int, y: Int) {
    let elementsPerRow = (image.width + UInt.bitWidth - 1) / UInt.bitWidth

    for row in 0..<image.height {
        let selfBaseOffset = self.bitIndex(x: x, y: y + row)
        let (selfIndex, selfOffset) = selfBaseOffset.quotientAndRemainder(dividingBy:
            BitImage.bitsPerElement)

        let otherBaseOffset = image.bitIndex(x: 0, y: row)
        let otherIndex = otherBaseOffset / UInt.bitWidth

        let secondWordShift = UInt.bitWidth - selfOffset

        for column in 0..<elementsPerRow {
            let imageBits = image.storage[otherIndex + column]

            self.storage[selfIndex + column] |= imageBits << selfOffset
            self.storage[selfIndex + column + 1] |= imageBits >> secondWordShift
        }
    }
}
}
```

4.2 Packing the Data for the GPU

Rasterising a lightmap using these scales and offsets is fairly straightforward; the clip-space position for each mesh vertex within the lightmap is given by the vertex's lightmap UV transformed by the offset and scale produced in the preprocessing stage. However, path tracing the lightmap is more complicated. Given

some position within the lightmap from which we wish to trace a ray, we need to be able to determine the instance and triangle index that overlaps that position within the lightmap. Once the instance and triangle indices are known we can generate a primary ray based on the vertex data for that triangle.

Barring some sort of lightmap-space acceleration structure, finding the mesh and triangle index amounts to a brute-force search through every triangle in the scene every time we want to generate a ray. We therefore need a better solution.

If, following Frostbite's implementation, we restrict ourselves to a fixed number of sample positions within each texel, we can precompute the triangle and instance indices for each of those sample positions. Generating a ray then becomes a matter of randomly choosing a valid sample position and looking up the instance and triangle index.

While this general approach is sound, it requires a massive amount of data; a 2048^2 texel lightmap with 32-bit triangle and mesh indices (implying $32 \times 2 \times 64 = 4096$ bits or 512 bytes per texel) leaves us with a 2GB lookup table. Quite aside from the fact that such a large table imposes immense bandwidth requirements on the GPU, the simple fact is that GPU memory is limited and a 2GB allocation has significant implications on what else can be stored in that memory.

We therefore need to somehow compress the data; conveniently, the data is highly amenable to compression. For most texels all of their sample points will be occupied by a single primitive, with only a few texels having a large number of primitives.

I propose encoding the data as follows. For every texel:

- Encode a 64-bit bitmask of the valid sample locations, where bit i is set if the i-th sample in the Hammersley sequence is valid within the texel.

- Then, for every triangle-instance pair, encode a 64-bit bitmask of the sample locations which that triangle-instance pair covers, followed by the 32-bit instance and 32-bit triangle index.

Since the amount of data required per texel varies, an offset buffer must also be generated such that if $j = \text{offsetBuffer}[i]$, $\text{sampleInfoBuffer}[j]$ contains the sample coverage information for texel i.

Using this method, the total memory usage for a 2048×1647 lightmap for the Sponza Atrium [5] model is 110.58MB, with 13.5MB of that being the offset buffer and the remaining 97MB being the per-sample information. This compressed result is 14.9 $\times$ smaller than the naïve implementation.

4.3 Decoding the Data on the GPU

The compressed format described in Section 4.2 can be decoded in a reasonably efficient manner on the GPU, although the implementation is complicated by the fact that 64-bit integer data types are often unavailable in GPU shading languages.

Given a target pixel location, each thread (responsible for generating one ray) retrieves the location in the sample information buffer from the offset buffer. Then, at that location, the first two 32-bit words (comprising on the 64-bit sample location bitmask) are retrieved. The number of valid samples is given by the sum of computing the `popcount` of each integer, where `popcount` is a built-in function that returns the number of non-zero bits in an integer.

An integer sample index i in the range $[0, \textit{validSampleCount})$ is then uniformly randomly generated. That index then needs to be mapped to the corresponding valid index j in $[0, \textit{sampleLocationsPerTexel})$, where *sampleLocationsPerTexel* is e.g. 64 if a 64-point Hammersley sequence [39] is used. To do this, we want to find the i^{th} set bit in the valid samples mask. If i is at least the number of non-zero bits in the lower 32-bit word, we want to find the $(i-$ `popcount` $(\textit{lowerWord}))^{th}$ set bit in the upper 32-bit word; otherwise, we want to find the i^{th} set bit in the lower 32-bit word.

Finding the n^{th} set bit in a word can be done without loops, making use of the `popcount` function and an eight-bit-indexed lookup table [66]. Alternatively, a loop that checks each bit in turn is a slower but simpler solution.

Once j has been found, we loop through all triangle-index pairs to find the first pair for which bit j is set in the valid-sample-location bitmask. Given that bit index, we can compute the sample location within the texel, reconstruct the barycentric coordinates based on the triangle's vertices' lightmap coordinates and the sample location, and from that reconstruct the world space position and normal that we need for generating the primary rays.

Listing 4.2 provides an implementation for the decoding. Note that it assumes that the word containing the lower 32 bits is stored before the word containing the upper 32 bits in a little-endian manner.

Listing 4.2: Lightmap Sample Information Decoding

```
uint texelSampleOffset = texelSampleBufferOffsets[y * uniforms.outputWidth + x];
uint validSamplesLowerBits = texelSampleBuffer[texelSampleOffset];
uint validSamplesUpperBits = texelSampleBuffer[texelSampleOffset + 1];
uint lowerBitsPopCount = popcount(validSamplesLowerBits);
uint upperBitsPopCount = popcount(validSamplesUpperBits);

uint validSampleCount = lowerBitsPopCount + upperBitsPopCount;

// Generate a sample from 0 to the valid sample count
uint sampleIndex = min(uint(validSampleCount * sampleGenerator.sample1D()),
    validSampleCount - 1);

// Find the bit index for the sample.
uint lookupInt;
uint bitIndex;
uint lookupOffset;
if (sampleIndex >= lowerBitsPopCount) {
   lookupInt = validSamplesUpperBits;
   bitIndex = sampleIndex - lowerBitsPopCount;
   lookupOffset = 1;
} else {
   lookupInt = validSamplesLowerBits;
   bitIndex = sampleIndex;
   lookupOffset = 0;
}

uint hammersleySampleMask = nthSetBit(lookupInt, bitIndex); // e.g. 0b1 for the 0th
    sample, 0b10 for the 1st sample etc.

uint texelSampleDetailsOffset = texelSampleOffset + 2;
while ((texelSampleBuffer[texelSampleDetailsOffset + lookupOffset] & hammersleySampleMask)
    == 0) { // Find the primitive index and shape index that corresponds to this sample
    index.
   texelSampleDetailsOffset += 4;
}

uint shapeIndex = texelSampleBuffer[texelSampleDetailsOffset + 2];
uint primitiveIndex = texelSampleBuffer[texelSampleDetailsOffset + 3];

// ctz == count trailing zeroes; e.g. given 0b100 the result would be 2
uint hammersleyIndex = ctz(hammersleySampleMask) + (sampleIndex >= lowerBitsPopCount ? 32
    : 0);
// Sample the Hammersely sequence using our sample index (and 64 samples per texel)
float2 sampleOffset = hammersleySample(hammersleyIndex, 64);
```

Chapter 5

Accelerating the Path Tracer: Improving Coherence

As discussed in Section 2.4, GPUs perform well on coherent workloads and poorly on incoherent ones. Given this, it is necessary for best performance (and therefore quickest artist iteration time) to arrange the workload for the GPU in a manner that provides coherent execution and memory access wherever possible. This chapter will consider stream compaction, a tile-based indexing method, and ray direction sorting as methods of improving performance by ensuring coherent workloads. Additionally, examples will be given of how the GPU radix sort implementations in prior work [56] can be accelerated using the SIMD operations available on recent hardware.

5.1 Stream Compaction

As overviewed in Section 2.4.2, a naïve path tracer where each thread operates on a single path is inefficient due to divergence within threadgroups. More concretely, not every path will be active in every bounce of the path tracer; many may miss geometry or be probabilistically terminated. The result of this is that in bounces after the first many threadgroups will only be partially occupied; since every thread in a threadgroup must perform instructions that any thread in the threadgroup performs (Section 2.4.1), this means that work is wasted on inactive paths.

Instead, to maximise coherence and throughput, it is worthwhile to perform *stream compaction* (Wald 2011) [53] to ensure that GPU threads operate only on active paths. To recap Section 2.4.2, stream com-

paction generates an index buffer containing only the indices of active paths, which is then used by later kernels to determine which path each thread should act on.

Stream compaction incurs its own overhead, which is amortised by the savings in the other kernels. In Wald's implementation [53] stream compaction was less than 1% of the total runtime; for the implementation in this thesis, we see stream compaction account for a more substantial 5% of the total runtime.

In LlamaEngine (Appendix B.1), following with the breadth-first split-kernel architecture of both RadeonProRender and Laine et. al. [11], the path tracers are split into many separate kernels to maximise coherence and throughput, and make use of stream compaction between those kernels to ensure full utilisation of the GPU's SIMD groups. Since the number of paths active at each stage is unknown by the CPU, the GPU writes the active path count to a buffer at various stages, which is then read by subsequent stages and used to terminate threads whose index is greater than the path count and therefore have no work to do.

The CPU can only specify the maximum number of threadgroups that may be required; if any paths have been compacted away, the actual number required is likely less. As the number of bounces increases and the number of paths drop there are an increasing number of threadgroups for which all threads terminate immediately after launching. This entails a significant overhead if a high number of bounces are desired (as might be necessary with highly specular paths). To counter this, I expand on prior work by simultaneously writing the required number of threadgroups for successive dispatches into indirect buffers in addition to the active path count. The GPU reads the number of threadgroups to dispatch from the current indirect buffer, eliminating the overhead of dispatching threadgroups which have no paths to work on and terminate early; this reduction in overhead is shown in Table 5.1.

While indirect buffers remove most of the overhead of dispatching zero-work threadgroups, sorting and compaction unfortunately have a fixed overhead determined by the maximum number of active paths. Since each threadgroup sorts or filters a fixed maximum amount of data and there is no direct ability to communicate between threadgroups, intermediate results must be written to device memory and merged in a separate dispatch. The number of dispatches for a sort must therefore be determined on the CPU based on the known maximum number of items.[1]

The performance impact is such that the per-bounce overhead when there are no active paths and

[1] As of June 2019, the Metal graphics API exposes the ability to enqueue compute dispatches directly from within compute shaders [49], which enables the theoretical removal of this fixed overhead; however, this capability was exposed after the work within this thesis and is therefore future work.

Table 5.1: Timing per frame for path tracing with and without indirect buffers at 2560×1440 resolution on Sponza Atrium (Figure D.5).

	Without Indirect Buffers	With Indirect Buffers	Percentage Improvement
12 bounces, RR after 4	$228ms$	$221ms$	3.2%
20 bounces, RR after 3	$210ms$	$192ms$	9.4%

As the number of inactive paths per bounce increases (either by lowering the first bounce at which Russian roulette is applied or by increasing the total number of bounces) the performance benefit of using indirect buffers increases.

with ray direction sorting enabled (see Section 5.4) is around $380\mu s$. Of this, all but $9.8\mu s$ is due to the sorting and compaction; the zero-count indirect dispatches have negligible overhead.

To incorporate stream compaction, the architecture from Section 3.1 is modified as follows, where alterations are *italicised*.

1. ***Fill an indirect buffer*** *with the threadgroup counts and active path count. If adaptive or lightmap rendering is not in use, this can be done by the CPU; otherwise, it is done while generating the path offset buffer (Section 3.1).*

2. **Generate primary rays.**

3. **Compute path radiances** using the path tracing estimator:

   (a) **Compute intersections.**

   (b) **Add radiance** *(modified)*: add any radiance *from any paths that missed scene geometry* to the *path* corresponding to each ray *and then mark those paths inactive*.

   (c) *In multiple dispatches,* ***filter and compact the path stream****, generating a buffer that contains the indices of all active paths. Note that this step differs from the RadeonProRender implementation since kernels have been combined together to reduce memory traffic. Section 2.5 on radix sort is a useful reference since many of the steps here are similar.*

      i. ***Count the number of active paths*** *within each threadgroup using SIMD operations and output that count to a buffer. Fill the indirect buffers with the threadgroup and active path counts.*

      ii. ***Perform a parallel prefix sum*** *over the per-threadgroup counts to compute the per-threadgroup offsets.*

      iii. *Compute the index i of each active path within its threadgroup and then* ***output the path index*** *to a buffer at the sum of i with the per-threadgroup offset. The path index is computed*

as a parallel prefix sum over all paths where the value for a thread is 1 if the path is active or 0 if it is not.

(d) **Generate indirect rays:** *for every thread index less than the active path count, fetch the corresponding ray and path indices*. For the rays that hit scene materials, generate a new ray starting at the surface and with a random output direction. Then, multiply the path throughputs by the surface BRDFs given the input and output directions.

4. **Accumulate samples.**

5.2 Tile-Based Indexing

Image pixels locations are two-dimensional; buffer indices are only one-dimensional. To store per-pixel information into a buffer we therefore need to map between 2D pixel locations and 1D indices. Examples of this per-pixel information include per-path data or ray information.

There are many possible invertible mappings from a 2D (x, y) output texel location to a unique integer given the dimensions of the output image. However, not every mapping is equal in performance; since the paths are sorted by the output of the mapping (Section 3.2), we ideally want the mapping to group together pixels whose paths are likely to be coherent. Additionally, we want the mapping to be fairly dense: we will be generating an offset buffer that maps from every possible output of the mapping to every input (Section 3.1), and large gaps in the mapping will leave inefficient gaps in the offset buffer.

Consider the trivial mapping of pixels in a row-major fashion, where:

$$f(x, y; \textbf{width}) = y \times width + x \tag{5.1}$$

Figure 5.1: Tile-based indexing with 4 × 4 tiles on an 8-wide grid

0	1	2	3	16	17	...
4	5	6	7	20	21	...
8	9	10	11	24	25	...
12	13	14	15	28	29	...
32	33	34	35	48	49	...
36	37	38	39	52	53	...
...	...	...	...	...	...	...

This satisfies the second constraint (i.e. it is a dense mapping), but is less than ideal for the first. In general, images are more similar in square blocks than in single-pixel-tall lines; a wide range of image encoding and compression algorithms exploit this fact [67]. In addition, in lightmap path tracing we ideally want to sort paths affecting the same triangle together since they are likely to be highly coherent; lightmap-space geometry tends to be closer to equilateral than highly stretched, making block-based sorting a good fit.

Given this, it is better to instead map from (x, y) pixel locations to locations within $n \times n$ *tiles*, where n is a power of two. A good choice for n is the square root of the number of threads per SIMD group to try to have one SIMD group per tile; for example, on AMD's GCN architecture $n = 8$. An index of this form is comprised of a *tile index* and *index within the tile*, which are bitshifted and bitwise OR'd together such that the tile index comprises the *bitWidth* $- \log_2(n)$ most significant bits and the index within the tile fills the lower $\log_2(n)$ bits. Let an index of this form be called a *tile format index*. Figure 5.1 shows an example of a tile-based indexing scheme.

If the dense mapping constraint is dropped, the mapping from a pixel coordinate to a tile format index $T(x, y)$ is given by:

$$T(x, y, \mathbf{n}, \textbf{tilesPerGrid}) = (\left\lfloor \frac{x}{n} \right\rfloor + \text{tilesPerGrid} \times \left\lfloor \frac{y}{n} \right\rfloor) << log_2(n^2) \\ \textbf{OR } ((x \textbf{ mod } n) + n \times (y \textbf{ mod } n)) \tag{5.2}$$

where $\text{tilesPerGrid} = \left\lceil \frac{width}{n} \right\rceil$, **OR** is a bitwise OR operation, and **mod** is the modulo operator. Since n is a power of two, the expensive modulo and division operations can be be replaced by bitwise operations; a **mod** n is equal to a **AND** $(n - 1)$ and $\left\lfloor \frac{a}{n} \right\rfloor$ is equal to $a >> \log_2(n)$.

The inverse mapping $T^{-1}(i)$ from a tile format index to a pixel coordinate is given by:

$$T^{-1}(i, \mathbf{n}, \textbf{tilesPerGrid}) = (n \cdot (i \textbf{ mod } \text{tilesPerGrid}), n \cdot \left\lfloor \frac{i}{\text{tilesPerGrid}} \right\rfloor) \tag{5.3}$$

In cases where the mapping from a given pixel to the first path that affects it is provided by the offset buffer this mapping is sufficient; there will be a few gaps in the offset buffer where there are incomplete tiles (tiles on the edge of the image where only some of the pixels lie within the dimensions of the output texture), but that is generally an acceptable tradeoff. If, however, there is no offset buffer, the paths are

packed densely into the paths buffer (e.g. in non-adaptive camera-based rendering where every pixel has exactly one path), and either the width or the height of the image is not divisible by n, we need a modified indexing scheme that ensures that there are no gaps in the mapped values for inputs within the image dimensions. In this case, $T(x, y)$ is defined as:

$$\begin{aligned} T(x, y, \mathbf{n}, \mathbf{tilesPerGrid}, \mathbf{width}, \mathbf{height}) =& (n \times \mathbf{min}(n, \text{height} - n \times \left\lfloor \frac{y}{n} \right\rfloor) \left\lfloor \frac{x}{n} \right\rfloor \\ &+ (n \times \text{width}) \left\lfloor \frac{y}{n} \right\rfloor \\ &+ (x \ \mathbf{mod}\ n) + (y \ \mathbf{mod}\ n) \times \mathbf{min}(n, \text{width} - n \times \left\lfloor \frac{x}{n} \right\rfloor) \end{aligned} \tag{5.4}$$

Note that tile-based indexing does not mean that each SIMD group can *only* operate on paths from within a single tile; fully-utilised SIMD groups are still preferred over coherent tiles. Having the data formatted in this way simply makes it more *likely* that SIMD groups will have coherent data, and is a straightforward way to achieve small performance gains.

Although I developed this indexing scheme independently, I noted upon implementation of the Metal Performance Shaders backend (Appendix B.3) that Apple recommends [58] a similar scheme, noting that such a scheme makes better use of GPU caches:

> When possible, organize rays within a batch for spatial locality. Rays that originate at nearby points or are oriented in similar directions tend to access the same locations in memory and can therefore make more effective use of the GPU's caches.
>
> For example, the camera rays associated with nearby pixels in the output image will likely originate at the same point and travel in very similar directions. Therefore, divide the output image into small tiles (e.g., 8×8). Rather than laying out all of the rays in the ray buffer in scanline order, first lay out the ray in scanline order within each tile, then lay out the tiles in scanline order or according to some space filling curve.

Across the scenes I have tested, tile-based indexing is a simple measure to achieve around 5% quicker render times; Table 5.2 shows the improvement for a specific case.

Table 5.2: Timing per frame for camera-based path tracing of Sponza Atrium (averaged over 1024 frames).

Linear row-based indexing	$191ms$
Tile-based indexing	$182ms$
Percentage improvement	4.9%

5.3 GPU Radix Sort with SIMD Operations

The core of my radix sort (Section 2.5) implementation is adapted from Harada and Howes' description in "Introduction to GPU Radix Sort" [56] and AMD's radix sort implementation within their Parallel Primitives library (which is bundled with RadeonProRender [43]). However, I have made some key adjustments to the implementation to optimise performance.

One useful property of performing multiple n-bit sorts is that knowing the range of possible values for your keys reduces the number of n-bit sort passes over the data required. In the GPU path tracer, it is true in many cases that the number of possible keys is restricted to some value; as such, I make use of this optimisation wherever possible. As an example, if you only consider values up to 512 (which makes use of the lower nine bits), only three passes must be performed rather than eight.

Within the implementation of the radix sort we can make use of lane-wise SIMD-group operations, which have existed in hardware for some time but have only recently been exposed by graphics APIs such as Metal [49]. Within a SIMD group, each thread can access local variables belonging to other threads within the same group without the need for synchronisation. This enables efficient execution of a number of parallel algorithms. Of particular interest to us for radix sort is the ability to efficiently perform a *parallel prefix sum* within a SIMD group: given N threads where each thread t has some value v_t, the value of $\sum_i^t v_i$ can be efficiently calculated. This capability was not available to the authors of AMD's Parallel Primitives library at the time it was written; however, I make heavy use of SIMD operations into my implementation.

More specifically:

Table 5.3: Timing for performing GPU radix sort on arrays of 32-bit numbers (averaged over 10 runs).

Element Count	Without SIMD	With SIMD	Percentage Improvement
$65,536$	$4.57ms$	$4.24ms$	7.8%
$64,000,000$	$738.56ms$	$599.66ms$	23.1%
$268,400,000$	$2656.84ms$	$2219.07ms$	19.7%

- Computing the count for each each key[2] within a threadgroup is done using a inclusive SIMD parallel prefix sum; after completion, the last thread within each SIMD group contains the total count across each SIMD group. To compute the total count within each threadgroup, the count from each SIMD group is copied into threadgroup memory, loaded by the first SIMD group, and then the same process is repeated.

- The parallel prefix sum over the threadgroup counts can be done using SIMD operations: each thread loads the count for one threadgroup from device memory, performs a parallel prefix sum within the SIMD group, and then outputs the result back to memory. Since this will only compute the prefix sum for as many threads as there are within a SIMD group (typically 32 or 64), this process needs to be recursively applied if the number of items for which to compute the parallel prefix sum is greater than the threads per SIMD group; this can be done by writing the count for each SIMD group to (either threadgroup or device) memory, loading those counts back into the SIMD registers for each thread, and then repeating the process.

- Determining the target offset of a key within a threadgroup (once the offset of that key within device memory for that threadgroup is known) can be done using a parallel prefix sum within that threadgroup.

Table 5.3 shows the performance improvements SIMD operations can provide in radix sort.

A parallel prefix sum can be implemented using only SIMD shuffles and additions. Starting with a stride of half the SIMD width and decreasing by half at each step, each thread adds the value contained in the thread at index *threadIndex* $-$ *stride*.[3] Figures 5.2 and 5.3 provide examples of SIMD summation.

Figure 5.2: SIMD Parallel Inclusive Prefix Sum: four threads

Action	Thread 0	Thread 1	Thread 2	Thread 3
Start Value	v_0	v_1	v_2	v_3
Add Thread (i - 2)	v_0	v_1	$v_0 + v_2$	$v_1 + v_3$
Add Thread (i - 1)	v_0	$v_0 + v_1$	$v_0 + v_1 + v_2$	$v_0 + v_1 + v_2 + v_3$

[2] From this point, 'key' will be used as shorthand for the n-bit section of each key that we are considering for the current sort pass.

[3] Alternatively, the Metal standard library provides an implementation in the form of `simd_prefix_exclusive_sum` or `simd_prefix_inclusive_sum` [49].

Figure 5.3: SIMD Parallel Inclusive Prefix Sum: eight threads with values

Action	Thread 0	Thread 1	Thread 2	Thread 3	Thread 4	Thread 5	Thread 6	Thread 7
Start Value	1	0	3	2	0	1	2	1
Add Thread (i - 4)	1	0	3	2	1	1	5	3
Add Thread (i - 2)	1	0	4	2	4	3	6	4
Add Thread (i - 1)	1	0	4	6	6	7	9	10

5.4 Ray Direction Sorting

One of the most expensive parts of the path tracing process is ray traversal and intersection testing. For ray traversal, we ideally want coherent rays: that is, rays that will be processed in the same SIMD group to have similar origins and directions and therefore take a similar path through the ray traversal acceleration structure and fetch cache-local data. Given this, it is sometimes worthwhile to sort the rays in some way; since traversal is a much more expensive operation than sorting, the cost of the sort is often amortised and performance improved.

In *Efficient ray sorting for the tracing of incoherent rays* (2012) [68], Nah et. al classify the rays into one of 24 direction buckets (determined by the primary direction and octant) and then stably sort by position, reporting a 1.48$\times$ speedup by doing so. I have also implemented ray direction sorting, although the implementation details differ due to the different structure of my path tracer. Additionally, in my implementation, I sort by the octant of the ray direction only. As described in Section 5.3, an eight-bit sort (as would be required to store 24 buckets) is double the cost of a four-bit sort, and in my tests the extra coherence from more refined bucketing did not offset the increased cost of the sort.

When ray sorting is enabled in my implementation, an additional buffer is passed to every shader stage that generates rays. When a ray is generated, the index of that ray within the rays buffer is bitwise ORed with the bucket index such that the bucket index is in the upper n bits and the lower $32 - n$ bits contain the ray index. A sort pass is then performed, operating only on the upper n bits; since radix sort is stable it will preserve the order of ray indices contained within the same bucket.

During the ray intersection kernels, the upper n bits are masked out and the ray for each thread is retrieved from the ray index specified by the lower bits. At the end of the intersection testing, the intersection results are placed back into the intersections buffer at the source ray index, ensuring that the source rays and the output intersections are in the same order.[4]

[4] This modification to the ray intersection kernels was not present in the original RadeonRays implementation (Appendix B.2).

Table 5.4: Frame timing breakdown for 'Modern Hall' (Figure D.4) [5] with Russian Roulette disabled, eight bounces, and ray direction sorting enabled at 2048 $\times$ 2048 resolution.

Stage	**Time**
Region Domain Generation	$0.21ms$
Primary Ray Generation	$0.78ms$
Paths Buffer Initialisation	$0.99ms$
Intersection Testing	$12.10ms$
Occlusion Testing	$8.82ms$
Path Stream Compaction	$7.08ms$
Surface Shading	$21.22ms$
Ray Direction Sorting	$4.48ms$
Light Sample Gathering	$2.99ms$
Sample Accumulation	$1.19ms$
Total	$59.86ms$

Modern Hall is a fairly geometrically simple scene, and as such surface shading is a larger portion of the total time (being slightly more expensive than intersection and occlusion testing combined). On scenes with more geometric complexity the balance shifts towards ray tracing being the most expensive stage.

Unfortunately, the performance argument for implementing ray direction sorting is fairly poor. On all of the scenes I tested, enabling ray direction sorting had effectively no impact on performance, the frame timings being so close as to be within the margin of error. This is not to say that the ray direction sorting is totally ineffectual, since if it were the frame timings would increase; rather, the improvements in ray intersection coherence are only enough to offset the cost of the sort. It may be more worthwhile on more geometrically complex scenes than are available to me, since, as Nah et. al. note, the performance improvement of ray direction sorting is generally "proportional to the number of ray-triangle intersection tests for each scene." Table 5.4 gives an example of the performance footprint of ray direction sorting relative to the full path-tracing process; if ray direction sorting is removed intersection and occlusion increase in time to make approximately the same total.

Ray direction sorting is only implemented within the RadeonRays intersector since I did not have access to the Metal Performance Shaders kernels to make the equivalent changes (Appendix B.2).

Chapter 6

Accelerating the Path Tracer: Reducing Variance

This chapter delves into methods of accelerating the path tracer by reducing the variance between samples, thereby improving the convergence rate and providing a better preview in less time. Next event estimation, the use of progressive sample sequences, and importance sampling are overviewed; for importance sampling, considerations when using a real-time layered material model are discussed. Additionally, rasterisation-based lighting methods that improve convergence at the cost of bias are discussed as alternatives to traditional path tracing lighting methods.

6.1 Next Event Estimation

Next event estimation is a commonly-used method that improves convergence in path tracing by explicitly sampling light sources. Traditionally, a path tracer recurses through the scene, adding radiance every time a light source is hit; therefore, at every intersection only an indirect ray is created. Next event estimation extends this by randomly selecting a known light source within the scene, calculating its contribution to the point being shaded, and then firing a shadow ray towards the light source to test for occlusion. To avoid double contribution from the lighting, the light's radiance must not be added directly if it is also hit by the indirect ray. Next event estimation is used by PBRT [13], RadeonProRender [43], and LuxRender [60], among others.

Since shadow rays only need to test for occlusion – whether there exists *any* intersection along each

ray, rather than finding the *closest* intersection along each ray – it is usually faster to perform occlusion queries for shadow rays than intersection queries for indirect rays.

Next event estimation is particularly important in the context of tracing scenes made for real-time applications such as games. Commonly, these real-time applications make use of analytic light sources with a delta distribution; point lights, spotlights (in the form of point lights with angularly varying emission), and directional lights do not exist in the real world and have no area. As such, there is no chance of an indirect ray being fired around the scene eventually hitting one of these light sources since they are infinitely small; instead, we must explicitly sample from them to account for their contribution.

Even when area lights are used, next event estimation remains useful since in many cases the probability of randomly hitting the area light is low. For instance, the sun has a small solid angle but extremely high intensity; explicitly sampling it ensures that we account for that intensity not only when a path randomly hits it, thereby reducing the variance.

Next event estimation requires the following modifications to our path tracing framework (Section 3.1), where changes are *italicised*:

1. **Generate primary rays.**

2. **Compute path radiances using the path tracing estimator:**

    (a) **Compute** (primary or indirect ray) **intersections.**

    (b) **Add radiance** at the intersection.

    (c) **Generate indirect rays:** for the rays that hit scene materials, generate a new ray starting at the surface and with a random output direction. *Additionally, sample a light source, output the path-throughput-multiplied BRDF-weighted contribution from that light source to a buffer, and generate a shadow ray to that light source.* Then, multiply the path throughputs by the surface BRDFs given the input and output directions.

    (d) ***Compute occlusion*** *of shadow rays with the scene.*

    (e) *For every light sample, check if the shadow ray missed the scene (i.e. hit the light). If the ray missed,* ***add the light's contribution*** *from the light samples buffer to the path's radiance.*

3. Accumulate samples.

6.2 Progressive Sample Sequences

Monte Carlo processes estimate a value by sampling. In the simplest approach, each sample is fully random, generated from entropy or approximated from some pseudo-random process. However, this yields poor convergence for Monte Carlo integration; since the sample distribution is random, the samples may not be well-distributed and could end up clumping in parts of the sample space.

Rather than utilising fully-random sampling, therefore, it is better to select samples from a sample set or sequence. An overview of the state-of-the-art is presented by Christensen, Kensler, and Kilpatrick. In the abstract to their paper *Progressive Multi-Jittered Sample Sequences* (2018) [69], they summarise:

> Sample patterns can be divided into two categories: finite, unordered sample sets, and infinite, ordered sample sequences. A progressive (a.k.a. hierarchical or extensible) sample sequence is a sequence where any prefix of the full sequence is well-distributed.
>
> Using (finite) sample sets requires a-priori knowledge of how many samples will be taken, and yields high error if only a subset of those samples are used. This is fine for rendering final images with a fixed number of samples per pixel. But in several common settings – including adaptive sampling – we do not know in advance how many samples will be taken, or we are using incremental results during computation as in interactive rendering and off-line rendering writing check-point images. In these cases we need (infinite) progressive sample sequences.

Infinite, ordered, progressive sample sequences are an ideal fit for progressive lightmap path tracing. In my implementation, I have included support for the Correlated Multi-Jittered (Kensler 2013) [70] and Sobol (Sobol 1967) [71] sample sequences, based on the implementation in AMD's RadeonProRender [43], along with Christensen et. al.'s PMJ and PMJ-02 sample sequences. These sampling sequences are compared in Figure 6.1.

To use a sample sequence, we need to keep track of the current sample index for every path. For non-adaptive camera-based sampling where every pixel receives one path per frame, this is simple: the sample index is just the frame index. For adaptive or lightmap rendering where the number of samples per pixel is probabilistic and variable, the sample index is retrieved via atomic operations on a pixel-indexed buffer. The pixel index is readily available when generating primary rays; therefore, at that stage the sample index is atomically read-and-incremented from the samples-per-pixel buffer and stored on the

`Ray`'s extra storage. At the first intersection, the sample index is transferred from the `Ray` to the `Path`, from which it is then read for the remaining bounces.

There are multiple different quantities to sample, and each should use its own sample sequence. The most important are primary ray samples (i.e. where on the camera lens to sample or what direction on the hemisphere to sample for lightmaps), light samples (for use in sampling area lights) and BRDF samples (for use in importance sampling the BRDF). Each of these quantities is known as a *dimension*.

Samplers can be seeded with a dimension and a sample index i to produce the i^{th} sample in that dimension. For Sobol and CMJ, these samples can be generated on the GPU, as is done by RadeonProRender. For the progressive multi-jittered sequences, however, and in particular for the PMJ-02 sequence, generation on the GPU is too costly. Instead, following Christensen et. al., I provide 128 precomputed 4096-element sequences in a buffer to the GPU.

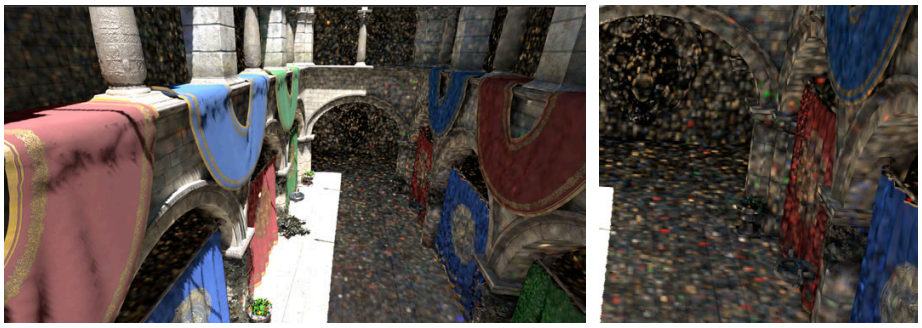
As O'Donnell notes [8], samples for neighbouring pixels should be decorrelated. If neighbouring texels share the same sampling sequences the lighting information will appear similar when other local information (such as normal or lightmap UV) is similar, resulting in visible discontinuities if some attribute becomes dissimilar (e.g. at lightmap seams). Instead, we should randomise the sequences used for each pixel, trading artefacts for noise.

To do this, a random buffer is used, which is initialised to a uniform random 32-bit integer for every pixel or texel at startup time. This random value can be used to determine the sample sequences to use. For example, by interpreting each 7-bit section of the integer as a sequence index (yielding up to four dimensions) each pixel receives a randomised combination of sequences. Even if neighbouring pixels do happen to receive the same sequence for one dimension, it is unlikely that they will receive the same sequence for another dimension, resulting in a decorrelated visual result. When sampling across multiple bounces or with more than four dimensions the random seed can be permuted in some way or the sequence indices can be incremented.

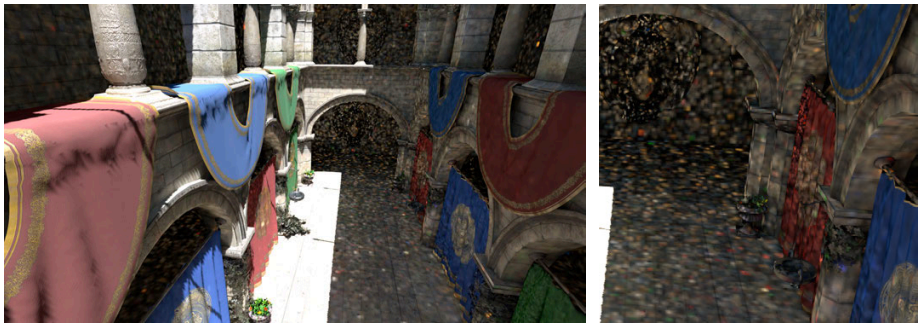
6.3 Importance Sampling

Importance sampling is a method to reduce the variance in Monte Carlo estimation by taking more samples in higher-value regions. Rather than generating samples uniformly within the sample domain (for example, over the hemisphere around the surface normal for a BRDF), samples are instead generated

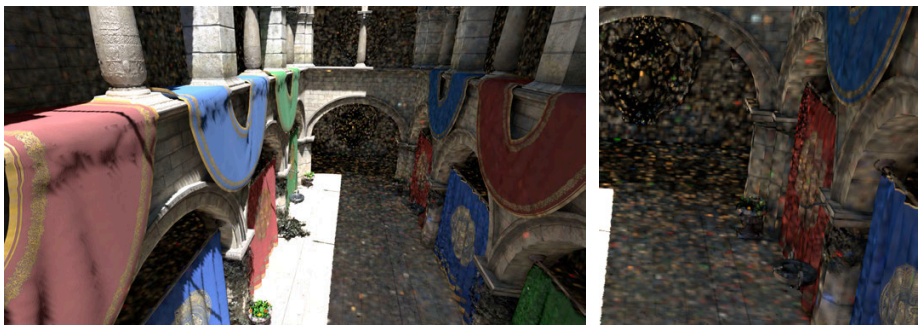
Figure 6.1: Comparison of sample sequences for sampling a camera-generated sample domain (Section 3.4.2).



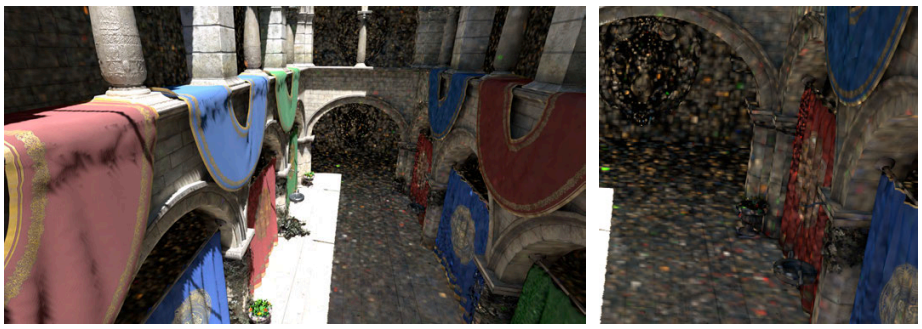
(a) Uniform random sampling



(b) Correlated Multi-Jittered sample sequence [70]



(c) Sobol sample sequence [71]



(d) PMJ-02 sample sequence [69]

518,400 paths were traced every frame, corresponding to a 960×540 *camera resolution, and eight frames were rendered.*
The CMJ, Sobol, and PMJ-02 sequences all perform relatively well, with uniform random sampling exhibiting significantly higher noise.

in proportion to their intensity; given a function $f(x)$ that we are trying to estimate, we want to generate samples for x such that we take more samples in areas where $f(x)$ is high-valued.

To ensure that the sampling remains unbiased, we must also divide by the probability PDF(x) of generating a sample with that x for every sample. The estimator simplifies in the case where we are able to exactly sample from the function distribution such that PDF(x) = $f(x)$. Consider an estimate of the product of two functions $f(x)$ and $g(x)$ where we importance sample $f(x)$:

$$\begin{aligned} \textit{Estimate} &= \frac{1}{n}\sum_{i=1}^{n} f(U_i)g(U_i) \\ &= \frac{1}{n}\sum_{i=1}^{n} \frac{f(s)g(s)}{\text{PDF}(s)} \\ &= \frac{1}{n}\sum_{i=1}^{n} \frac{f(s)g(s)}{f(s)} \\ &= \frac{1}{n}\sum_{i=1}^{n} g(s) \end{aligned} \tag{6.1}$$

The *probability distribution function* PDF(x) is the derivative of the inverse of the sampling distribution. The integral of the PDF is known as the *cumulative distribution function* or CDF; when generating a sample x with a probability PDF(x) from a uniform random sample U, x will be given by CDF$^{-1}(U)$.

Importance sampling is critical in achieving a low-variance result in path tracing and should be applied wherever possible. Although importance sampling can be used anywhere that sampling is required, there are two main areas for which it is most useful: sampling of materials and of lights.

6.3.1 Importance Sampling Materials

In path tracing, we randomly choose a new ray direction at every bounce after hitting a surface and then multiply the throughput of the path by the surface's BSDF in that direction. Rather than uniformly randomly choosing a new direction, we can in many cases generate directions by importance sampling the BSDF, thereby reducing variance.

Although the list of BSDFs in use is growing increasingly long as real-time applications shift to more complex material models, there remain two main BSDFs of interest for many applications: Lambertian diffuse and GGX specular. Both can be efficiently importance sampled.

For Lambertian diffuse, sampling in a cosine-weighted hemisphere around the surface normal is sufficient; this can be done by uniformly randomly generating points on a disk and then projecting those samples onto a hemisphere. Since the height of a point on the hemisphere is determined by the cosine of the angle between the normal and the point (i.e. $z = cos(\theta)$) this will result in cosine-weighted samples.

For the GGX distribution, Heitz introduced an efficient method in *Sampling the GGX Distribution of Visible Normals* (2018) [72]. While this method only importance samples one component of the GGX specular function (excluding the visibility and Fresnel components), the variance reduction is significant and the method can be efficiently implemented on the GPU without any need for precomputation or lookup tables.

Sampling Real-Time Material Models

One goal of a lightmap baking tool for real-time applications is to try to match the appearance within the real-time application as closely as possible so that the lighting and materials designed for the rasteriser match the artist's intentions when used within the path tracer. In effect, the rasteriser becomes an approximate ground truth; if the lighting baked by the path tracer does not match with the models within the rasteriser, even if the path tracer produces a more physically correct result, the path tracer can be considered incorrect.

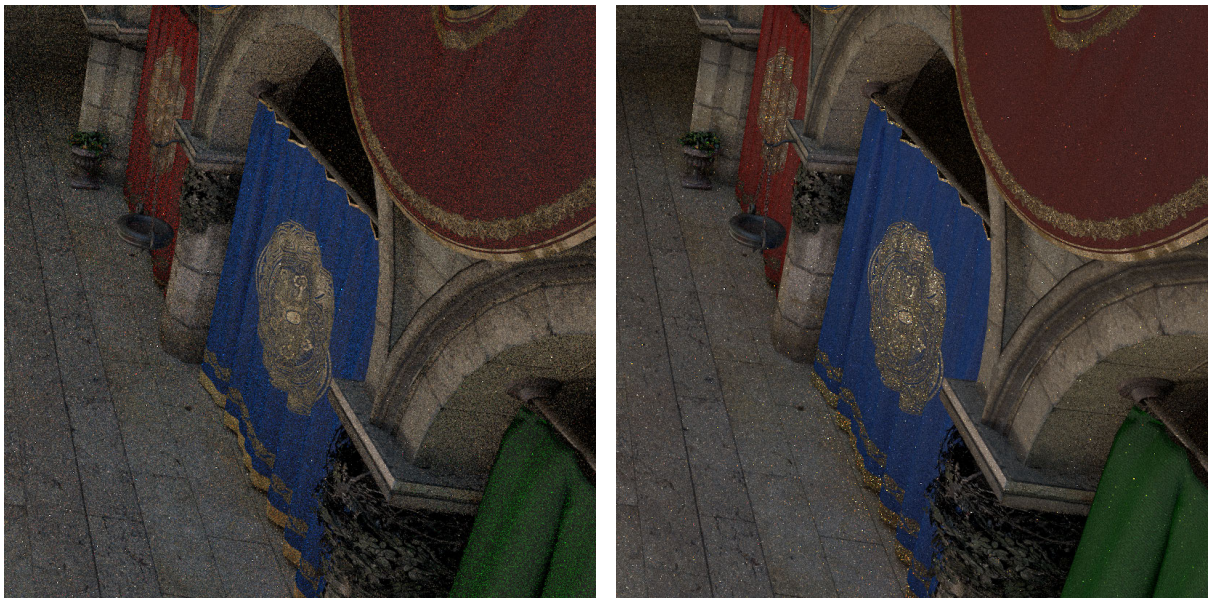
One way in which path tracers and real-time applications commonly differ is in their use of material models. In particular, layered materials are very difficult to accurately simulate in a rasteriser but can be modelled much more simply in a path tracer.

The most common layered material used in real-time applications such as games is an additive diffuse and specular mix. Such a model is not necessarily energy conservative. In many cases, the diffuse and specular layers are separately normalised to obtain an approximately energy conserving response for a range of roughness values and then simply added together during evaluation; this is the approach taken by LlamaEngine (Appendix B.1).[1]

In an additive model, we still want to be able to importance sample the BRDF. We can do this by randomly choosing a layer to sample – either diffuse or specular, for example – and then dividing that layer's radiance by the probability of choosing that layer. Given a probability p of sampling the specular layer instead of the diffuse:

[1] A more accurate method is to model the Fresnel reflectance and transmission for the specular layer, computing how much energy gets transmitted to and reflected out of the diffuse layer for each view direction, roughness parameter, and light direction. Recent variants of such a method are provided by Weidlich and Wilkie [73] and Jakob et. al [74].

Figure 6.2: Evaluating path traced material layers separately vs. simultaneously.



(a) Evaluating separately

(b) Evaluating simultaneously

On the left, a material layer is selected to be sampled, the intensity is divided by the selection probability, and then that layer is evaluated and divided by the sampling PDF. On the right, a material layer is selected, no modification is made to the intensity, and then all layers are evaluated and divided by the sampling PDF. Note the 'fireflies' in the centre symbol on the blue curtain when both layers are evaluated simultaneously; this is caused by high-intensity low-probability paths.

$$
\begin{aligned}
radiance &= \frac{1}{N}\sum_{N}(diffuse + specular) \\
&= \frac{1}{N}(\sum_{N} diffuse + \sum_{N} specular) \\
&\approx \frac{1}{N}(\frac{1}{1-p}\sum_{(1-p)N} diffuse + \frac{1}{p}\sum_{pN} specular)
\end{aligned}
\tag{6.2}
$$

Alternatively, it is possible to importance sample from one layer and then evaluate for all layers; however, doing so may result in low probability paths for the importance-sampled layer having high intensities for the other layer, causing high-intensity 'fireflies' to appear in the final image (Figure 6.2).

Regardless of how indirect rays are sampled, direct light sampling with next event estimation should, where possible, always evaluate every layer in the same way that a light would be evaluated within a rasteriser.

6.3.2 Importance Sampling Lighting

In next event estimation, a light must be chosen to be explicitly sampled at every surface interaction. In my implementation, light sampling is performed by randomly choosing a light with probability proportional to its intensity, omitting any spatial considerations.

This method was chosen solely due to its simplicity, and there are many lower variance, more sophisticated methods for choosing which lights to sample from when shading a particular hit point. Two recent techniques are *Adaptive Direct Illumination Sampling* (Vévoda and Kþivánek, 2016) [75] and *Importance sampling of many lights with adaptive tree splitting* (Estevez and Kulla, 2017) [76], both of which build complex hierarchical structures. Estevez and Kulla also use sampling heuristics weighing the orientation, energy, a BRDF approximation, and the distance between the hit point and the light. In scenes with a large number of lights where an unbiased result is desired, a solution such as Estevez and Kulla's would be greatly beneficial in reducing the sampling variance.

Area lights (including emissive surfaces and the environment map) deserve particular attention. For these types of lights, next event estimation is not strictly necessary; since they have some area, given enough samples a path will always hit any contributing area light. However, the convergence rate of this can be poor, particularly for rough surfaces with a wide BRDF lobe. To compensate for this, we can use a strategy called *multiple importance sampling* (Veach and Guibas, 1995) [77]. Multiple importance sampling (MIS) combines multiple sampling strategies – for example, combining BRDF importance sampling with explicit light sampling based on next event estimation – by weighting the contribution from each sampling strategy according to some heuristic based on the probability of each sampling strategy for the given sample. One commonly used heuristic is the *balance heuristic*, where the weight w_i for sampling strategy i is given by:

$$w_i(s) = \frac{\text{PDF}_i(s)}{\sum_j \text{PDF}_j(s)} \tag{6.3}$$

In the context of light sampling, the process is to first sample the light in a direction determined by the light, weight by the balance heuristic, and add the contribution to the path's radiance; and then sample the light in a direction determined by the BRDF, weight by the balance heuristic, and add that contribution to the path's radiance. While theoretically straightforward, this has one major complication in the context of a breadth-first GPU path tracer.

To determine a light's contribution in a particular direction, we first need to know whether the light

is occluded in that direction. To check whether a light is occluded, we either use a shadow ray (for light sampling) or an indirect ray (for BRDF sampling); only when processing the indirect ray's intersection do we know whether it hit the light. If the intersection is with something other than the area light it means that the area light's contribution in the BSDF direction was zero, and we do not need to handle this in any special way; however, if the intersection is with the area light then we need to weight its contribution by the balance heuristic before adding it to the path's accumulated radiance.[2] This necessitates keeping track of the light sampled at the previous step and the weight w_{bsdf} for importance sampling the light using the BSDF in some memory corresponding to each ray.

6.4 Biased Light Sampling

If a biased result is acceptable, there are other alternatives to direct lighting than path-traced next event estimation that are easily integrated into many existing 3D engines. One approach is to forgo ray-traced shadows and instead use the standard shadow mapping techniques implemented within the rasteriser, thus enabling many or all lights to be sampled at each hit point.[3] This has the additional benefit of exactly following the rasteriser's lighting path, giving a biased result that is nevertheless a closer match to the final rasterised image.

Additionally, it is possible to reuse existing algorithms for use in light culling or spatial sampling. In recent years, lighting techniques for rasterisation have favoured tiled or clustered shading [78, 79, 79, 80]. Usually implemented in view-space, tiled shading divides the output image into a series of small tiles and generates a list of lights for each to sample from; clustered shading extends this with a series of depth slices, forming a 3D grid-like structure. Both techniques assume a finite falloff radius for each light's intensity, which does not match the actual inverse square falloff and is therefore biased; however, in practice there is a point at which a light's contributions are negligible and therefore can be discarded.

View-space clustered shading can be trivially adapted to build a world-space grid for clustered shading in path tracing. To do so, an orthographic frustum is tightly fit to the scene bounds along one of the major axes, forming an axis-aligned bounding box. When shading a hit point, the hit point's position is

[2] If we intersect with an area light that is not the light we explicitly sampled at the previous step we add its contribution without any MIS weighting.

[3] With ray-traced shadows, one shadow ray must be generated and traced for every light that is sampled, which quickly becomes untenable in the case of many lights. The ideal number of lights to sample at each hit point is a tradeoff between convergence rate and time per sample that varies from scene to scene.

transformed into the space of that bounding box, from which the 3D cluster cell can be found. Lighting can then be performed using the lights specified for that cluster, either by using all lights with their associated shadow maps or by uniformly randomly sampling one of the lights affecting the cell.

Biased light sampling in this manner fully supports all BRDFs and light types that are supported by the rasteriser, making it a good option when baking radiance-encoding lightmaps (rather than lightmaps encoding only diffuse irradiance) for real-time scenes. In the case of area lights where no accurate, efficient solution for shadowing exists within the rasterisation-based pipeline, Heitz, Hill, and McGuire present a technique that allows analytic evaluation of area lights in conjunction with accurate ray traced shadows [81]. Such a technique could equally be employed here, wherein punctual and directional lights use rasterised shadow maps and other light types are given accurate shadows by way of ray tracing.

6.4.1 Irradiance Caching

Irradiance caching is a family of techniques that aim to cache the direct lighting within a scene. Introduced in 1998 by Ward, Clear, and Rubinstein [82], and extended by numerous authors in a SIGGRAPH 2007 course [83]), the basic concept is that the rate of convergence of path tracing can be greatly improved by caching lighting information in a spatial data structure in a preprocessing pass, albeit at the cost of a biased result.

In the context of a GPU-focused lightmap path tracer, the simplest implementation is an irradiance cache storing direct Lambertian irradiance as a single colour value for each texel.[4] This can be achieved by simply rasterising the scene to a lightmap, optionally making use of the world-space clustered shading grid to accelerate lighting. Then, during path tracing, the incident lighting is retrieved from the lightmap texture.

Rasterising the scene to a lightmap can be performed very quickly, and retrieving the lighting from the lightmap is also very inexpensive. Convergence time is greatly increased since every vertex along a path will contribute radiance from *every* light, and the path tracing process is quicker due to the lack of shadow rays. Figure 6.3 demonstrates the improved convergence rate irradiance caching provides.

Irradiance caching comes with a few limitations, however. First and foremost is that the BRDFs in the scene are restricted to be Lambertian:[5] any view dependency is lost when storing as a scalar texture.

[4] This is the approach used by EA's Frostbite engine [4].

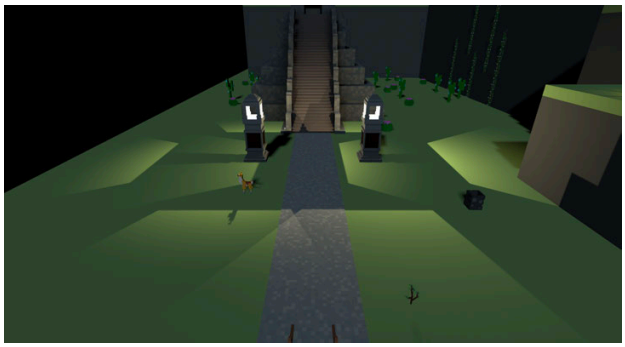
[5] There exist closely related more sophisticated techniques under the name of *radiance caching*, which captures directionality and may also capture multiple bounces of light.

This can be alleviated somewhat by using spherical basis functions to encode the distribution of the exitant lighting; see Křivánek et al. [84] for details. However, note that illumination due to direct lighting with glossy BRDFs is generally high frequency, and is therefore difficult to accurately represent with most spherical basis functions; for example, all of the spherical basis functions discussed in Chapter 7 preserve only low-frequency information.

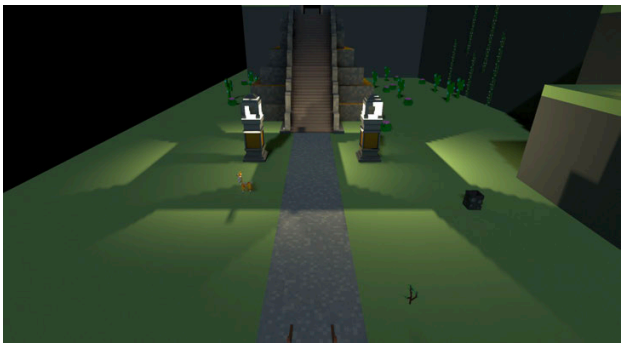
Secondly, the lighting information is limited by the resolution of the lightmap: sharp cutoffs or shadows lose definition and are blurred. If the material albedo is baked into the irradiance cache there is also a corresponding loss in texture detail; however, it is also possible to apply the material albedo for each hit point during path tracing, since for Lambertian irradiance the albedo linearly scales the BRDF.

Care must also be taken when rasterising the lightmap. The output position for each mesh vertex is given by its lightmap UVs; however, it is possible that some triangles may not be rasterised to the lightmap since they do not cover the pixel centre. When sampling from the lightmap, this leads to black patches where there should be valid irradiance values. To alleviate this, conservative rasterisation should be used where supported, ensuring that triangles which overlap a pixel in any way will always be rendered. If conservative rasterisation is unsupported (either by the hardware or by the rendering API), standard antialiasing techniques can be used in conjunction with a dilation filter that fills empty pixels with the blended contents of neighbouring filled pixels.

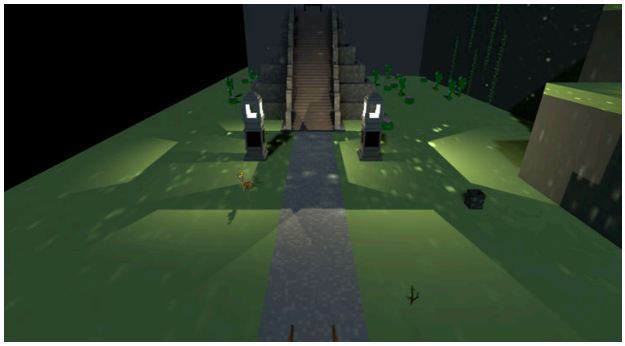
Figure 6.3: Irradiance Caching



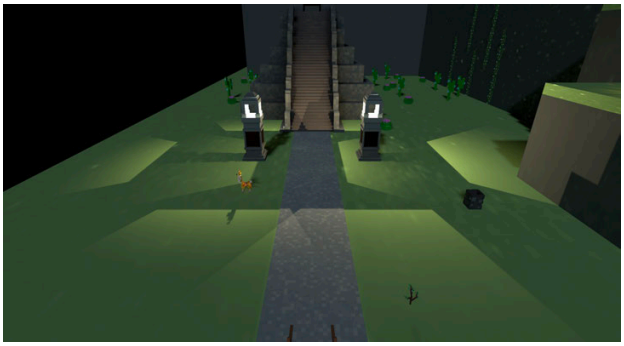
(a) Rasterised scene without indirect lighting.



(b) Path traced scene using a rasterised direct illumination lightmap (irradiance cache) for irradiance lookups.



(c) Rasterised scene with indirect from a path-traced scalar irradiance lightmap, rendered without irradiance caching.



(d) Rasterised scene with indirect from a path-traced scalar irradiance lightmap, rendered with irradiance caching.

This scene from Interdimensional Llama (see Appendix B.1) in many ways represents a best-case scenario for irradiance caching: many lights, complex shadowing, and diffuse surfaces.

All images were rendered for 10 seconds. Note the marked decrease in visible noise when using the lightmap rendered with irradiance caching compared to that rendered without. Without irradiance caching, many light samples are occluded or low-intensity, resulting in slow convergence; however, irradiance caching enables the lighting environment to be efficiently captured.

Chapter 7

Spherical Basis Functions

In previous chapters, we have mainly been concerned with computing the radiance within the scene. This chapter shifts the focus onto how that radiance can be stored in a manner that improves the visual quality of the reconstructed lighting.

As discussed in Section 2.2, it is in many cases more useful to store the lightmap's incident lighting in a format other than a simple colour value. Doing so enables support for normal mapping and convolution with various arbitrary BRDFs; for example, reconstructing diffuse and specular indirect lighting from a single representation.

One such format is a linear spherical or hemispherical basis. Depending on the choice of basis function, encoding into a linear basis may or may not be trivial. In this chapter, the mathematical formalism behind encoding into linear bases is described, with a focus on techniques that require little storage or computation and are therefore suitable for the GPU. Additionally, a new method is presented that enables non-negative least-squares encoding to be performed progressively on the GPU.

7.1 Linear Bases

It is often useful to approximate a function $f(s)$ by some linear combination of basis functions $B(s)$. In computer graphics, for example, we may want to approximate an infinitely-high-frequency incident light distribution with some combination of basis functions, reducing the storage required to store that light distribution to a single coefficient for each basis function.

Given some target function $f(s)$, a set of basis functions $B_i(s)$ (often called *lobes* if the basis functions

are of the same type in different directions), and a set of weights b_i for each of those basis functions, we can approximate the function value in direction s as:

$$f(s) \approx \sum_i b_i B_i(s) \tag{7.1}$$

A simple example of this is a polynomial basis where $B_i(s) = s^i$, where a truncated Taylor series approximates the function. Other commonly used bases include the Fourier basis, where a function is approximated as the sum of sine waves of varying frequencies, and Bernstein polynomials, used in Bézier curves and piecewise splines.

If the basis function is defined such that $B(s) = B(||s||)$ (where $||s||$ is the absolute value of s) the basis function is known as a radial basis function. This is particularly useful for spherical functions, where the basis function can be defined such that it depends on the angular distance from some point on the sphere. Figure 7.1 shows an example of lighting reconstructed from a linear basis of radial basis functions, while the individual contribution of each function is shown in Figure 7.2.

When encoding into a linear basis, the target function $f(s)$ can either be known analytically or sampled using Monte Carlo integration. The basis functions must be fixed for a given encoding, although they may be arbitrary and unrelated to each other.

In the case of lightmaps, the indirect lighting can be stored into a set of textures, where each texture represents one coefficient b_i at various points within the scene.

Figure 7.1: Approximation of the Wells HDR environment map [85] with the Ambient Dice SRBF basis functions.

$f(s)$ $\sum_i b_i B_i(s)$

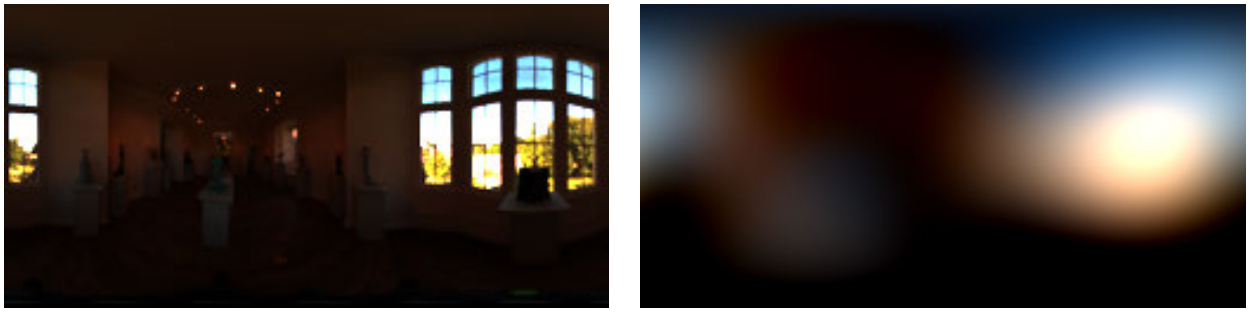
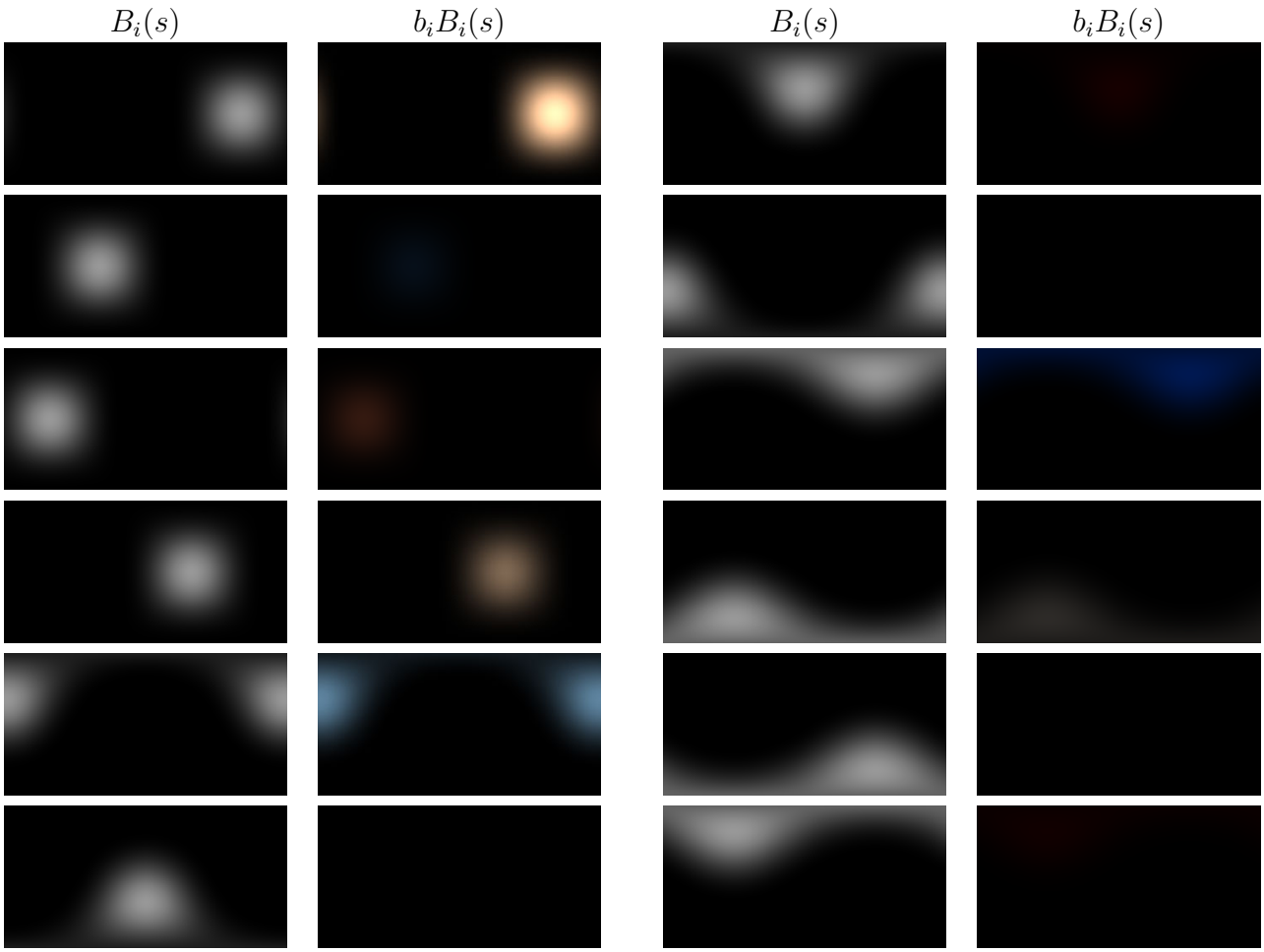


Figure 7.2: The individual Ambient Dice SRBF basis functions comprising the approximated Wells HDR environment map.



7.2 Least-Squares Encoding of Spherical Basis Functions

To approximate $f(s)$ with some linear combination of basis functions $\sum_i b_i B_i(s)$ we need to find the weight vector b. Since, in general, $f(s)$ may not be exactly represented by the linear basis, we instead need to minimise the error according to some metric.

One such metric is the least-squares error, or the squared difference between the true value and the approximation. The least-squares error may be easily solved for and provides good quality results, although it does disproportionately weight outliers and is not necessarily ideal in all cases.

Minimising the least-squares error can be done in a functional-analysis manner by solving the following equation:

$$\min \int_S (\sum_i b_i B_i(s) - f(s))^2 \, \mathrm{d}s \tag{7.2}$$

where S is the integration domain; typically this will be over the entire sphere, but the same techniques apply over the hemisphere or over other arbitrary domains.

To minimise, we differentiate the function with respect to each unknown b_i and then set the derivative to 0.

$$E = \int_S (\sum_i b_i B_i(s) - f(s))^2 \, \mathrm{d}s$$
$$\frac{dE}{b_i} = 0$$

Let $g(s) = \sum_j b_j B_j(s) - f(s)$. $\frac{d}{b_j}\Big[g(s)\Big] = B_j(s)$ for each b_j.

$$
\begin{aligned}
\frac{dE}{b_i} &= \frac{d}{b_i}\left[\int_S (\textstyle\sum_i b_i B_i(s) - f(s))^2 \,\mathrm{d}s\right] \\
&= \frac{d}{b_i}\left[\int_S (g(s))^2 \,\mathrm{d}s\right] \\
&= 2\int_S g(s)\frac{d}{b_i}\left[g(s)\right]\mathrm{d}s \\
&= 2\int_S g(s)B_i(s)\,\mathrm{d}s \\
&= 2(\sum_j b_j \int_S (B_i(s)\cdot B_j(s)))\,\mathrm{d}s - 2\int_S (B_i(s)\cdot f(s))\,\mathrm{d}s
\end{aligned}
$$

Therefore, by setting $\frac{dE}{b_i} = 0$,

$$
\sum_j b_j \int_S (B_i(s)\cdot B_j(s)))\,\mathrm{d}s = \int_S (B_i(s)\cdot f(s))\,\mathrm{d}s \tag{7.3}
$$

This is a standard least-squares matrix equation. On the right-hand side we have the *raw moments*, which, when performing Monte Carlo integration, are the projection of the sample values onto the basis functions[1]; on the left, we have the weight vector b multiplied by the Gram matrix, where $G_{ij} = \int_S (B_i(s)\cdot B_j(s))\,\mathrm{d}s$. To find the weight vector b we can multiply the raw moments by the inverse of the Gram matrix:

$$
\begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = \begin{pmatrix} \int_S (B_1(s)\cdot B_1(s)) & \int_S (B_1(s)\cdot B_2(s)) & \dots & (B_1(s)\cdot B_n(s)) \\ \int_S (B_2(s)\cdot B_1(s)) & \int_S (B_2(s)\cdot B_2(s)) & \dots & (B_2(s)\cdot B_n(s)) \\ \vdots & \vdots & \ddots & \vdots \\ \int_S (B_n(s)\cdot B_1(s)) & \int_S (B_n(s)\cdot B_2(s)) & \dots & (B_n(s)\cdot B_n(s)) \end{pmatrix}^{-1} \begin{bmatrix} \int_S (B_1(s)\cdot f(s)) \\ \int_S (B_2(s)\cdot f(s)) \\ \vdots \\ \int_S (B_n(s)\cdot f(s)) \end{bmatrix}
$$

If the raw moments are computed by Monte Carlo integration, the Gram matrix can be applied to each sample to perform progressive encoding. Given a set of samples where the k^{th} sample s_k a value $f(s_k)$, the weight vector b is given by:

$$
b = \frac{1}{k}\sum_k f(s_k)(G^{-1}B(s_k))
$$

[1] Projecting the sample values onto the basis functions simply means multiplying each sample value $f(s)$ by $B(s)$ and accumulating the result.

The Gram matrix is the identity matrix for sets of orthonormal basis functions such as spherical harmonics (section 8.1); therefore, performing a least-squares solve for spherical harmonics requires only projecting the radiance samples onto the basis functions and does not need a matrix multiplication. However, many basis functions, such as spherical Gaussians or Ambient Dice, are not orthonormal, and for these basis functions standard least-squares methods are necessary.

The functional analysis method of least-squares solves was brought to my attention by Peter-Pike Sloan, who also published a brief description of the method in *Ambient Dice* (Iwanicki and Sloan, 2018) [12]. In the context of sampling and Monte Carlo integration, a functional-analysis style solve is actually a non-obvious approach. The most immediate option, given a set of samples and a set of basis functions, is to construct a matrix system that directly tries to solve for the lobe amplitudes; i.e. given a set of radiance samples s_k with values $f(s_k)$, the matrix system would be:

$$\begin{pmatrix} B_1(s_1) & B_2(s_1) & \dots & B_n(s_1) \\ B_1(s_2) & B_2(s_2) & \dots & B_n(s_2) \\ \vdots & \vdots & \ddots & \vdots \\ B_1(s_n) & B_2(s_n) & \dots & B_n(s_n) \end{pmatrix} \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = \begin{bmatrix} f(s_1) \\ f(s_2) \\ \vdots \\ f(s_n) \end{bmatrix}$$

In prior work, the absence of a functional-analysis approach has presented challenges. Since the obvious method of least-squares optimisation is performed on all samples, that requires having all sample values and directions be stored in memory and means that progressive rendering is impossible.

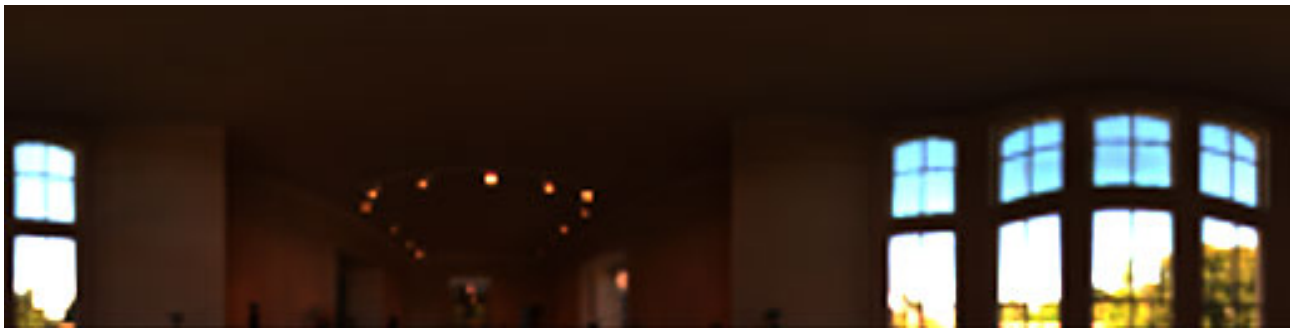
For example, Pettineo and Neubelt made use of spherical Gaussians in *The Order: 1886* [31]. Facing the problems of expensive least-squares solves and high memory costs, they decided that a naïve solution would have to suffice: they simply projected the samples onto each lobe independently as if each one formed an orthonormal basis [10].[2] The results from this method are visually poor compared to a least-squares or non-negative least-squares solve, appearing washed out and lacking definition.

7.2.1 Spherical Basis Functions Over the Hemisphere

In most cases, spherical basis functions are solved to minimise the error over a spherical domain; for example, if the function represents incident light at a point, the approximation should account for the light in all directions.

[2] Unbeknownst to them at the time, this is in fact exactly the raw moments in the functional analysis solution; they were one matrix multiplication away from the full reconstruction.

Figure 7.3: Spherical basis functions over the hemisphere on the Wells HDR environment map [85].



(a) The upper hemisphere of the Wells HDR environment map.

(b) Ambient Dice Cosine-Lobe SRBF over the sphere (RMSE: 0.557).

(c) Ambient Dice Cosine-Lobe SRBF over the hemisphere (RMSE: 0.522).

The hemisphere-constrained solve yields a much closer visual match to the source image, although the mean-squared error is only slightly reduced.

However, in some cases, it may appear to make sense to minimise the error only over a hemisphere. For example, in lightmap baking the incident lighting is zero for all directions in the hemisphere opposite the surface normal, and all directions to be queried will be on the hemisphere. Minimising the hemispherical error will yield higher quality results on that hemisphere for radiance reconstruction since all information below the hemisphere can be discarded (Figure 7.3).

The mathematical formalism is identical for least-squares solves over any domain, including over the sphere and hemsiphere. When generating the Gram matrix for the hemisphere by sampling, sample directions should be taken from the uniform hemisphere about the surface normal – usually defined to be $(0, 0, 1)$ in tangent space – rather than from a uniform sphere.

Unfortunately, solving over the hemisphere means that every lobe has a different total influence (i.e. $\int_\Omega B_i(\omega)^2 \, \mathrm{d}\omega$), rather than the same influence in different directions. This can cause issues when using fits to try to approximate the cosine-weighted irradiance or specular lighting from each lobe since each lobe needs to be weighted differently.

To elaborate, if the basis functions are defined over the hemisphere, there is an implicit guarantee that sampling in any direction below the hemisphere will result in zero radiance. While this may seem obvious, consider that spherical basis functions are often used to enable normal mapping, wherein the shading normal differs from the geometric normal (and therefore the centre of the hemisphere). In the case where the shading normal is not aligned with the geometric normal, care must be taken in integrating the BRDF to ensure that all directions below the *geometric* normal's hemisphere return zero radiance.

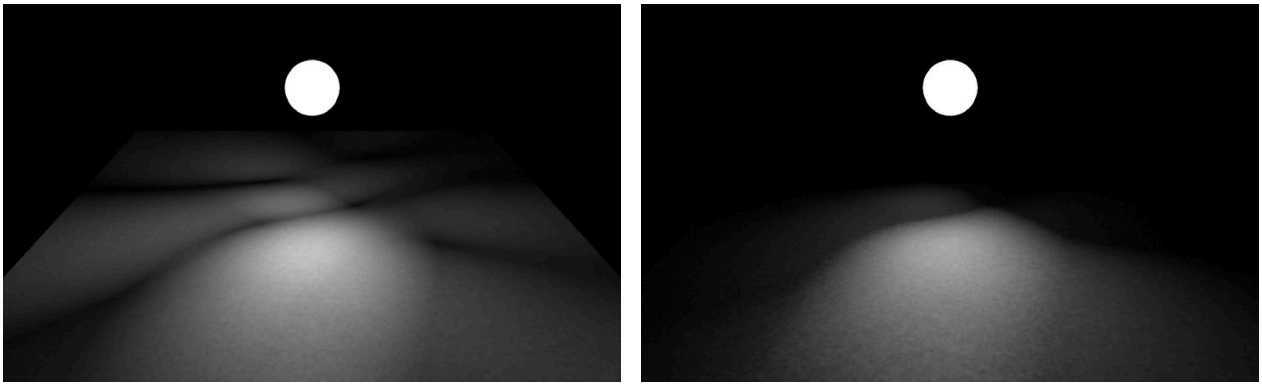
This issue makes finding analytic solutions to the BRDF integral challenging, since the integral must be clipped by the geometric hemisphere, the basis function's domain (if it is not the entire sphere), and the BRDF's hemisphere. In comparison, basis functions defined on the sphere only need to account for clipping by the BRDF's hemisphere and the basis function's domain (if applicable).

7.3 Progressive Least-Squares Encoding

Both the functional analysis and direct least-squares solves have a few shortcomings, particularly in the context of progressive encoding. For progressive encoding – as is desired when viewing the results of a progressive path-trace of a lightmap – we need to have a valid b vector for use in reconstruction at every step so that we can display the partial result of the solve with the samples taken so far.

A direct solve is impractical in this context, which means a functional analysis-style solve must be used. When the basis functions are not orthonormal, this necessitates an $N \times N$ matrix multiplication at each step (where N is the number of lobes), which in turn requires $N \times N$ fused multiply-add instructions to simply calculate the weights. Both least-squares solve methods also pose no constraints upon the b vector, meaning that the per-lobe weights can become negative. This can, in turn, make the reconstructed radiance negative in parts of the function space. Negative light is physically implausible, and is particularly problematic when approximating specular lobes, as can be seen in Figure 7.4.

Figure 7.4: Indirect specular from spherical Gaussian lightmaps.



(a) Least-Squares **(b)** Non-Negative Least-Squares

In this section I propose a novel method for progressive least-squares encoding for spherical basis functions (Listing 7.3.1).[3] This method is efficient, can run on the GPU, converges fairly quickly, and requires storing only the current amplitude b_i and a weight for each lobe. Crucially, it allows the imposition of arbitrary constraints upon the lobe amplitudes by simply projecting the constraints onto the values at each iteration of the algorithm; for non-negative encoding, for example, that entails clamping the basis amplitudes to be strictly positive at each step.

The conceptual underpinning of the method to try to evaluate how accurately the function $R(s) = \sum_i b_i B_i(s)$ approximates each incoming radiance sample $f(s)$ and to adjust each lobe amplitude b_i by the difference in a form of gradient descent. More formally, it is a special case of Jacobi or Gauss-Seidel iteration for when the function space is iteratively sampled.

To provide background context, it is first useful to look at how you might solve the system (either for the accumulated raw moments or for each sample) using Jacobi or Gauss-Seidel iteration, two iterative algorithms for solving systems of linear equations in a least-squares manner. At each step, Jacobi iteration

[3] This technique was first informally published in a blog post based on research done as part of this thesis [86].

applies the following method to solve the equation $Ax = b$, where A is a matrix and x and b are vectors:

$$x_i^{(k+1)} = \frac{b_i - \sum_{j \neq i} A_{ij} x_j^{(k)}}{A_{ii}} \tag{7.4}$$

Gauss-Seidel iteration differs only in that it updates each element of the x vector in turn and uses the updated elements to calculate the rest; Jacobi iteration will compute the entirety of the $x^{(k+1)}$ vector before overwriting any part of the previous $x^{(k)}$ vector.

If we apply the Jacobi algorithm to our functional analysis least-squares equation $Gb = m$, where m is the vector of moments (the accumulated projection of the sample values onto the basis functions), we get:

$$b_i^{(k+1)} = \frac{m_i - \sum_{j \neq i} G_{ij} x_j^{(k)}}{G_{ii}} \tag{7.5}$$

$$= \frac{\int_S (B_i(s) \cdot f(s))\,\mathrm{d}s - \sum_{j \neq i} \int_S (B_i(s) B_j(s)) b_j^{(k)}\,\mathrm{d}s}{\int_S B_i(s)^2\,\mathrm{d}s} \tag{7.6}$$

Similarly, it is possible to apply Jacobi or Gauss-Seidel iteration per-sample rather than on the whole system, yielding:

$$b_i^{(k+1)} = \frac{B_i(\omega)(f(\omega) - \sum_{j \neq i} B_j(\omega) b_j^{(k)})}{B_i(\omega)^2} \tag{7.7}$$

On their own, neither per-sample nor full-system Jacobi or Gauss-Seidel is particularly useful for progressive least squares encoding in linear bases. The full-system variant requires that we have all samples in advance, which makes it non-progressive, while the per-sample variant produces an unusably noisy estimate to the b vector for the sample's ω.

However, we can combine the per-sample and full-system variants to create a new method which is ideal for progressive solves. In particular, if we take the numerator from the per-sample version and the denominator from the full system, we can, with a few alterations, perform progressive least-squares encoding. To derive the method, we start back at Equation 7.3 and solve for a single b_i, assuming that all b_j are known from a prior iteration:

$$\begin{aligned}\int_S (B_i(s) \cdot f(s))\,\mathrm{d}s &= \sum_j b_j \int_S (B_i(s) \cdot B_j(s))\,\mathrm{d}s \\ &= b_i \int_S B_i(s)^2\,\mathrm{d}s + \sum_{j,j\neq i} b_j \int_S (B_i(s) \cdot B_j(s))\,\mathrm{d}s\end{aligned}$$

$$b_i \int_S B_i(s)^2\,\mathrm{d}s = \int_S (B_i(s) \cdot f(s))\,\mathrm{d}s - \sum_{j,j\neq i} b_j \int_S (B_i(s) \cdot B_j(s))\,\mathrm{d}s \tag{7.8}$$

We can bring the entire right hand side under the same integral due to the linearity of integration.

$$\begin{aligned}b_i \int_S B_i(s)^2 &= \int_S (B_i(s) \cdot f(s) - \sum_{j,j\neq i} b_j (B_i(s) \cdot B_j(s)))\,\mathrm{d}s \\ &= \int_S (B_i(s) \cdot (f(s) - \sum_{j,j\neq i} b_j \cdot B_j(s)))\,\mathrm{d}s\end{aligned}$$

Finally, we end up with the following equation for b_i:

$$b_i = \frac{\int_S (B_i(s) \cdot (f(s) - \sum_{j,j\neq i} b_j \cdot B_j(s)))\,\mathrm{d}s}{\int_S B_i(s)^2\,\mathrm{d}s} \tag{7.9}$$

There are two integrals here that can be computed iteratively using Monte Carlo integration. The denominator, $\int_S B_i(s)^2\,\mathrm{d}s$, can be precomputed; however, it is more accurate in practice to instead compute the denominator in lockstep with the numerator since that helps to cancel out sampling bias. At every step of the algorithm, the denominator is stored separately from the b vector, requiring $C+1$ values per lobe where C is the number of colour channels.

Note that Equation 7.9 is very similar to Equation 7.6 for Jacobi iteration on the whole system, with the key difference being that the entire numerator is brought under the same integral. In addition, rather than performing Jacobi iteration on the vector of raw moments m, we instead perform one Jacobi or Gauss-Seidel iteration *per-sample* and compute the final b vector using a Monte Carlo process, where the b_i value at each iteration is derived from the b vector in the previous iteration.

Using that iterative framework, we can simplify the numerator by introducing and factoring out $b_i^{(k)}$:

$$
\begin{aligned}
b_i^{(k+1)} &= \frac{\int_S (B_i(s) \cdot (f(s) - \sum_{j,j \neq i} b_j^{(k)} \cdot B_k(s)))\,\mathrm{d}s}{\int_S B_i(s)^2\,\mathrm{d}s} \\
&= \frac{\int_S (B_i(s) \cdot (f(s) - \sum_j b_j^{(k)} \cdot B_j(s) + b_i^{(k)} \cdot B_i(s))\,\mathrm{d}s}{\int_S B_i(s)^2\,\mathrm{d}s} \\
&= \frac{\int_S (B_i(s) \cdot (f(s) - \sum_j b_j^{(k)} \cdot B_j(s))\,\mathrm{d}s + b_i^{(k)} \int_S (B_i(s)^2)\,\mathrm{d}s}{\int_S B_i(s)^2\,\mathrm{d}s} \\
&\approx \frac{\int_S (B_i(s) \cdot (f(s) - \sum_j b_j^{(k)} \cdot B_j(s))\,\mathrm{d}s}{\int_S B_i(s)^2\,\mathrm{d}s} + b_i^{(k)}
\end{aligned}
$$

Let $\Delta = f(s) - \sum_j b_j \cdot B_j(s)$, or the difference between the current sample value and the current estimate for the current sample's direction. Δ is constant for all lobes within a given a particular sample and therefore only needs to be computed once per iteration. Therefore, for each i, we can compute the b_i estimate for a particular sample in direction ω_s as:

$$
\mathit{Est}_s(b_i^{(k+1)}) = \frac{B_i(\omega_s) \cdot \Delta_s}{\int_S B_i(s)^2\,\mathrm{d}s} + b_i^{(k)} \tag{7.10}
$$

Welford's algorithm [57] is a numerically stable algorithm for computing the mean and variance of some sample set. At each step, the mean μ is updated with a new sample s as follows:

$$
\mu^{(k+1)} = \mu^{(k)} + \frac{s - \mu^{(k)}}{k} \tag{7.11}
$$

Therefore, to accumulate the various Monte Carlo estimates for b (which is simply a matter of averaging the estimates for each sample given uniform random sampling), we can apply:

$$
b_i^{(k+1)} = b_i^{(k)} + \frac{1}{k}\left(\frac{B_i(\omega_s) \cdot \Delta_s}{\int_S B_i(s)^2\,\mathrm{d}s} + b_i^{(k)} - b_i^{(k)}\right) \tag{7.12}
$$

$$
= b_i^{(k)} + \frac{1}{k}\left(\frac{B_i(\omega_s) \cdot \Delta_s}{\int_S B_i(s)^2\,\mathrm{d}s}\right) \tag{7.13}
$$

This method will iteratively converge to the least-squares solution for b. The speed of its convergence depends on the sample distribution, the initial estimates, and an *acceleration factor* α. It turns out to be possible to increase the convergence rate at the method at the cost of increased visible noise during the

solve (since the solution will overshoot and correct itself) by performing, at each step:

$$b_i^{(k+1)} = b_i^{(k)} + \frac{\alpha}{k}\left(\frac{B_i(\omega_s) \cdot \Delta_s}{\int_S B_i(s)^2 \,\mathrm{d}s}\right) \tag{7.14}$$

A reasonable range for α is between 1 and 5. In my tests, I found $\alpha = 3$ to provide the quickest convergence in a range of scenarios.

7.3.1 Notes and Limitations

This technique is conditional upon the distribution of the incoming radiance samples. If the sample directions are uniformly randomly distributed then the result will converge to the minimum mean-squared error; however, if the sample directions are highly correlated the result will be very poor. Fortunately, we naturally want the sampling pattern to be uncorrelated in most contexts where we are accumulating radiance samples progressively; in path tracing, for instance, stratified sampling is often used to ensure that successive samples are not over-representative of a particular direction. Note also that progressive sample *sequences* that converge quickly within the first few samples are ideal (Section 6.2), while sample *sets* such as Hammersley [39] that only cover the sample space once all samples have been taken are very poor choices.

It is also conditional upon the convergence of Jacobi or Gauss-Seidel iteration given a particular basis function. Jacobi iteration is known to converge when the system is diagonally dominant; in this case, that means:

$$\int_S B_i(s)^2 \,\mathrm{d}s > \sum_{j,j\neq i} \left|\int_S B_i(s) B_j(s) \,\mathrm{d}s\right|$$

although it may also converge under other conditions. Gauss-Seidel iteration, on the other hand, will converge in any case where the Gram matrix is symmetric and positive definite, which will always be the case if the basis functions are strictly positive-valued. Under both Jacobi and Gauss-Seidel iteration a more diagonally dominant matrix will converge quicker than a less diagonally dominant one. More practically speaking, the greater overlap there is between the basis functions the slower the system will converge; very wide spherical Gaussian lobes, for example, cause issues for this method. In practice, in these cases the method will often achieve a reasonably good result but then asymptotically decrease the rate of convergence, requiring a very high sample count to fully converge.

To use Gauss-Seidel rather than Jacobi iteration within this encoding method we need to update the value of Δ after solving for every lobe:

$$\begin{aligned}
\Delta_{i+1}^{(k+1)} &= \Delta_i^{(k+1)} + b_i^{(k)} B_i(\omega) - b_i^{(k+1)} B_i(\omega) \\
&= \Delta_i^{(k+1)} + (b_i^{(k+1)} - \frac{\alpha}{k}(\frac{B_i(\omega) \cdot \Delta_i^{(k+1)}}{\int_S B_i(s)^2 \,\mathrm{d}s}))B_i(\omega) - b_i^{(k+1)} B_i(\omega) \\
&= \Delta_i^{(k+1)} - \frac{\alpha}{k}(\frac{B_i(\omega) \cdot \Delta_i^{(k+1)}}{\int_S B_i(s)^2 \,\mathrm{d}s})B_i(\omega) \\
&= \Delta_i^{(k+1)}(1 - \frac{\alpha}{k}\frac{B_i(\omega)^2}{\int_S B_i(s)^2 \,\mathrm{d}s})
\end{aligned}$$

Given those conditions and using Gauss-Seidel iteration this algorithm will always converge to the least-squares solution in the unconstrained case. As a rough heuristic, this progressive least-squares algorithm exhibits similar error to performing somewhere between five and eight iterations of the Gauss-Seidel algorithm on the full system set up in a functional-analysis least-squares manner (i.e. $Gb = m$, where G is the Gram matrix, b is the weight vector, and m is the vector of projected moments). This is true regardless of whether Gauss-Seidel or Jacobi iteration is used within the algorithm; however, this algorithm will only converge using Jacobi iteration in situations where Jacobi iteration on the full system would converge.

Extra constraints may be introduced by projecting the basis amplitude onto those constraints after every iteration; for example, a non-negative solve can be achieved by clamping the amplitude to be non-negative after each sample is added. Doing so may prevent the system from ever reaching the true value and has no formal mathematical basis, although intuitively you can reason that subsequent samples will compensate for the constraint in their solve. In practice, the results from non-negative clamping come very close to those achieved using a dedicated non-negative solver on the full system.

Special care must be taken in evaluating the denominator $I = \int_S B_i(s)^2 \,\mathrm{d}s$. As already mentioned, the denominator should be computed in lockstep with the numerator; as each basis function's weight $B_i(\omega)$ is evaluated, the value of I should be updated to be the average of all $B_i(\omega)^2$ values encountered thus far. However, for low sample counts, the b estimate will be very noisy, and, since the range of B_i is often $[0, 1]$ for many basis functions, noise in the estimate can be greatly amplified by noise in I.

I have found two effective methods to mitigate this. The first is to clamp the value used in the denominator for calculating b_i to at least the true value of I; however, this requires precomputing I. The second

method, which has slightly lower error on my test sets, is to interpolate from the value of I used in the naïve projection – that is, 1 – to the true value based on the sample index k:

$$I_i^{(k)} = \frac{1}{k} + (1 - \frac{1}{k}) \cdot \frac{1}{k} \sum_{j=1}^{k} B_i(\omega_j)^2 \tag{7.15}$$

If the incoming sample directions are defined uniformly over the hemisphere (as is the case in lightmap baking) but the integration domain should be over the sphere, additional samples should be added after each true sample with a direction opposite the upper hemisphere direction and a radiance value of zero.

Listing 7.1: Progressive Least-Squares Encoding

```
float lobeMCSphericalIntegrals[lobeCount] = 0.f;
float totalSampleWeight = 0.f;

for sample in radianceSamples {
 totalSampleWeight += sample.weight;
 float sampleWeightScale = 1.f / totalSampleWeight;
 Colour delta = sample.value;
 float sampleLobeWeights[lobeCount];

 for lobeIndex in 0..<lobeCount {
  float weight = lobes[lobeIndex].evaluateBasis(sample.direction);
  delta -= lobes[lobeIndex].amplitude * weight;
  sampleLobeWeights[lobeIt] = weight;
 }

 for lobeIndex in 0..<lobeCount {
  float weight = sampleLobeWeights[lobeIndex];
  float sphericalIntegralGuess = weight * weight;
  lobeMCSphericalIntegrals[i] += (sphericalIntegralGuess - lobeMCSphericalIntegrals[i]) *
      sampleWeightScale;

  float basisSphericalIntegral = sampleWeightScale + (1.f - sampleWeightScale *
      lobeMCSphericalIntegrals[i]);
  float deltaScale = acceleration * weight * sampleWeightScale / basisSphericalIntegral;
  lobes[lobeIndex].amplitude += delta * deltaScale;

  if nonNegativeSolve {
   lobes[lobeIndex].amplitude = max(lobes[lobeIndex].amplitude, Colour(0));
  }

  // If we want to perform Gauss-Seidel iteration:
  delta *= 1.0 - deltaScale * weight;
 }
}
```

7.3.2 Implementation

This method has been implemented and tested across a range of different software on both the CPU and GPU. Initial prototyping was done within the open source tool Probulator [87], and was later tested within the open source tool The Baking Lab [18]; the GPU implementation was tested within LlamaEngine (Appendix B.1).

For the GPU implementation, samples were traced from locations in a lightmap and were accumulated into a 32-bit float RGBA render target per basis function, with the spherical integral I stored in the alpha channels. A single-channel 32-bit float render target was also used to store the total accumulated sample weight since the sample count varies per texel and results are splatted across multiple texels using filtered weights.

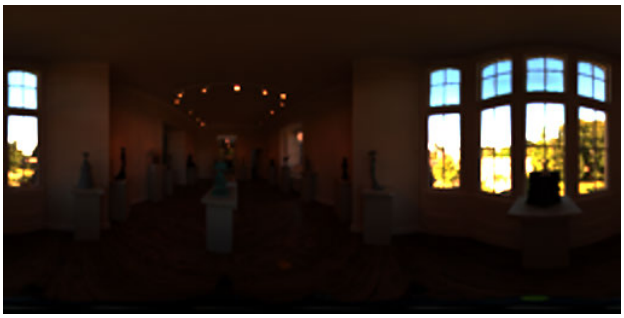
In lightmap tracing contexts, the current lobe amplitudes should be point-sampled for the texel being solved for. Although it may seem more correct to linearly filter the lobe amplitudes when the ray originates near an edge or corner of the texel, doing so causes lobes to use their neighbours to minimise the error, resulting in a noisy checkerboard-like pattern.

Care must be taken in regards to the storage of the intermediate weight vector b. In particular, 16-bit floating point is insufficiently precise to capture the minute adjustments to the weights and will cause biasing towards large sample values. In my implementation, all intermediate results were stored in 32-bit floating point; preliminary tests done with 64-bit floating point showed minimal improvement in accuracy over 32-bit.

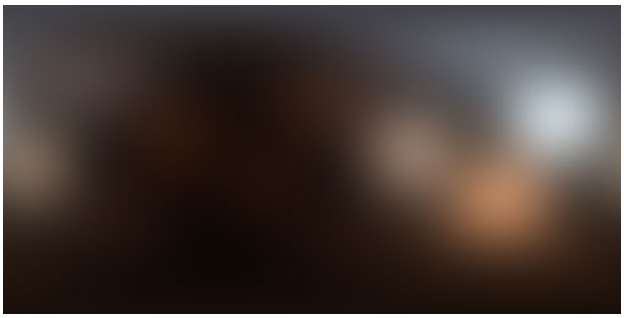
7.3.3 Results

Figure 7.5 shows a comparison of the naïve, functional-analysis least-squares, and progressive least--squares encoding methods, showing the close match achieved by progressive-least-squares. Figure 7.6 shows how encoding quality is negatively impacted by the use of a sample set rather than sequence, while Figure 7.7 demonstrates the close proximity of the progressive non-negative solve to the true value. An example of convergence compared with the least-squares solution is given in Figure 7.8.

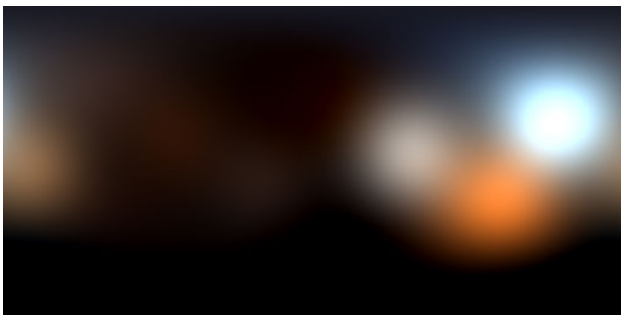
Figure 7.5: Comparison of the naïve projection, least-squares, and progressive least-squares encoding methods.



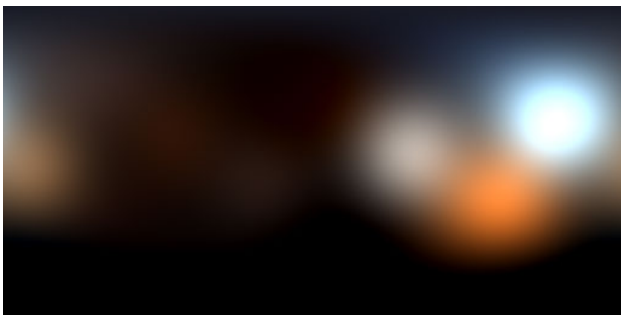
(a) The Wells HDR environment map [85].



(b) Naïve projection onto 12 spherical Gaussian lobes (RMSE: 0.470525).

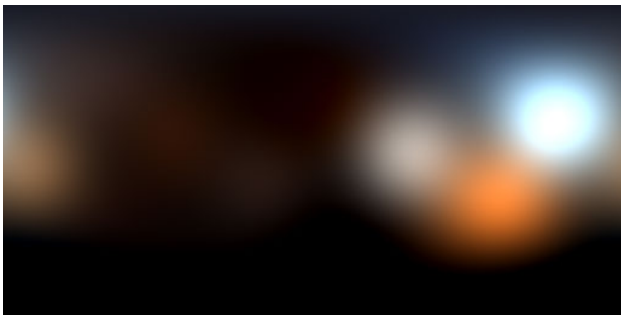


(c) A least-squares fit with 12 spherical Gaussian lobes (RMSE: 0.455862).



(d) Progressive least-squares encoding using 12 spherical Gaussian lobes (RMSE: 0.455971).

Figure 7.6: Progressive least-squares encoding with correlated vs. decorrelated samples.



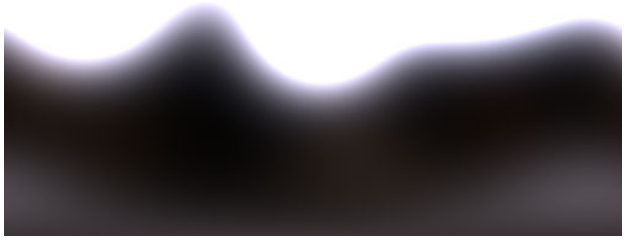
(a) Progressive least-squares encoding with samples from the Halton 2-3 sequence.

(b) Progressive least-squares encoding with correlated samples from the Hammersley set [39].

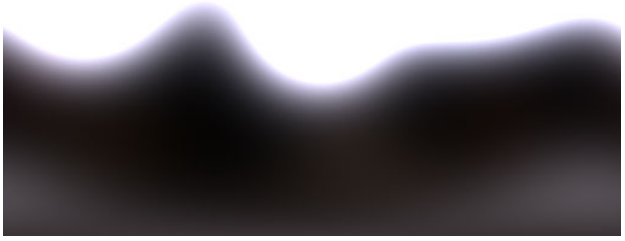
Figure 7.7: Progressive least-squares encoding with negative vs. non-negative lobe amplitudes.

(a) The Uffizi HDR environment map [88].

(b) A least-squares fit using 12 spherical Gaussian lobes (RMSE: 3.07549).

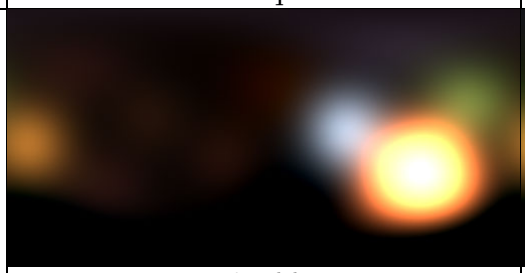
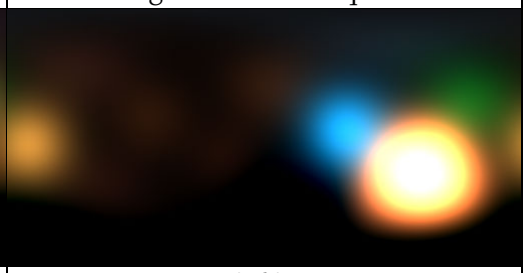
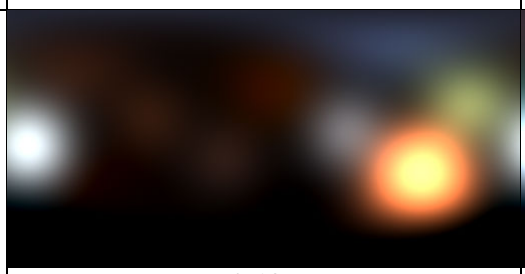
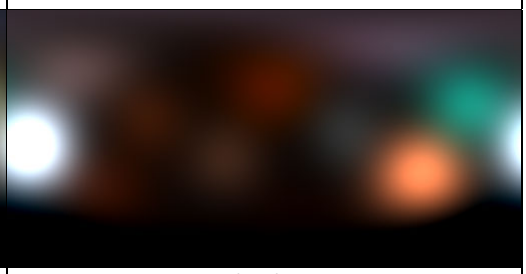
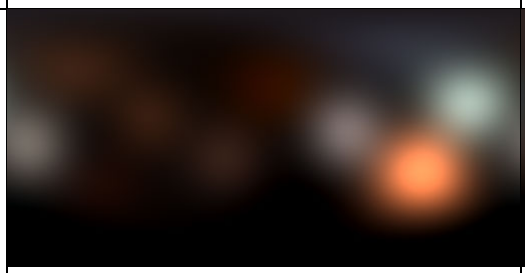
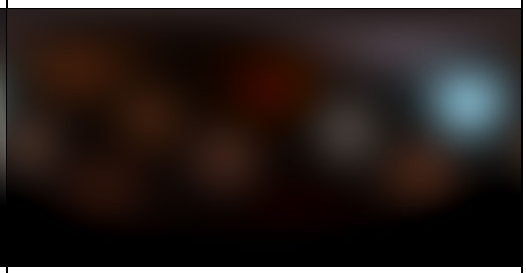
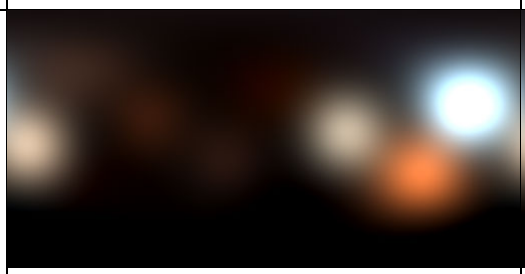
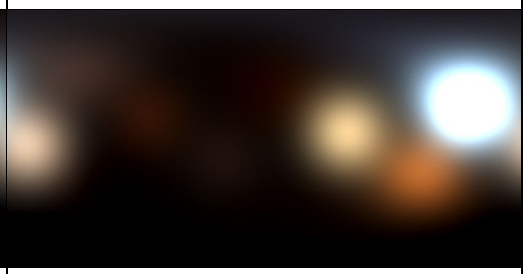
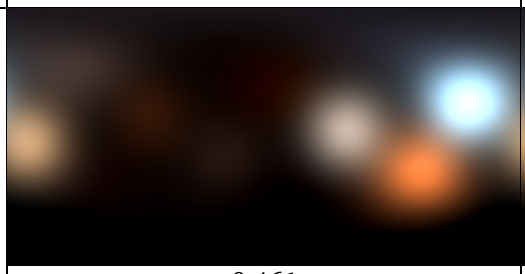
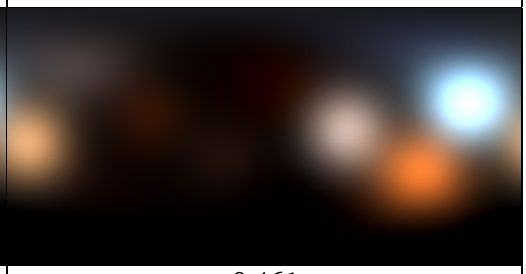


(c) A non-negative least-squares fit using 12 spherical Gaussian lobes (RMSE: 3.13181).



(d) Progressive non-negative least-squares encoding using 12 spherical Gaussian lobes (RMSE: 3.13928).

Figure 7.8: Convergence rate of the progressive least-squares encoding method.

	Least-Squares	Progressive Least-Squares
32 Samples		
RMSE	0.566	0.601
64 Samples		
RMSE	0.495	0.507
128 Samples		
RMSE	0.472	0.493
256 Samples		
RMSE	0.462	0.472
512 Samples		
RMSE	0.461	0.461

Wells HDR environment map [85] with twelve spherical Gaussian lobes ($\lambda = 8$)

7.4 Encoding BRDF-Weighted Spherical Basis Functions from Radiance Signals

In addition to encoding radiance directly into linear bases, it is also possible to encode a BRDF-convolved signal. For example, you may wish to have a basis function store cosine-weighted irradiance rather than radiance so that you can directly query the irradiance. This is particularly useful for basis functions such as Ambient Dice (Section 8.3) [12], which have local support (that is, different directions fetch different sets of coefficients).

One method of doing this is to project the radiance signal into some intermediate basis in which the convolution can be efficiently performed. For example, in *Ambient Dice* Iwanicki and Sloan project the radiance signal into high-order spherical harmonics, in which cosine-lobe convolution can be efficiently performed, and then encode the convolved signal into the final basis function. As they note, "For very fine functions this is somewhat impractical, since a very high order SH expansion would have to be used." This section will show how this general method – projecting into one basis, convolving with the BRDF, and then projecting into the final basis – can be applied to any spherical basis function, and not just spherical harmonics.

First, note that a signal, represented as a linear combination of basis functions, can be projected into some new space by a matrix multiplication of the amplitudes of the basis functions in the original space. Practically speaking, this matrix multiplication can be folded directly into the solve by multiplying with the inverse Gram matrix.

Consider Equation 7.2 where the function value $f(s)$ is the convolution of $R(\omega)$ and a symmetric function[4] $C(\omega_i, \omega_o)$:

$$\min \int_S \left(\sum_i b_i B_i(s) - \int_S (R(\omega) \cdot C(\omega, s)) \, \mathrm{d}\omega \right)^2 \mathrm{d}s \tag{7.16}$$

Let $R(s)$ be the radiance signal represented by some linear combination of basis functions:

$$R(\omega) = \sum_k a_k A_k(\omega) \tag{7.17}$$

We can then reformulate the least-squares equation as:

[4] A symmetric function here means a function for which $C(\omega_i, \omega_o)$ has the same value as $C(\omega_o, \omega_i)$.

$$\min \int_S \left(\sum_i b_i B_i(s) - \sum_k a_k \int_S (C(\omega, s) A_k(\omega)) \, \mathrm{d}\omega \right)^2 \mathrm{d}s \tag{7.18}$$

Solving this equation in the same manner as before yields:

$$\sum_j b_j \int_S (B_i(s) \cdot B_j(s)) \, \mathrm{d}s = \sum_k a_k \int_S (B_i(s) \cdot \int_S (C(\omega, s) A_k(\omega)) \, \mathrm{d}\omega) \, \mathrm{d}s \tag{7.19}$$

On the left we have our familiar Gram matrix $G_{ij} = \int_S (B_i(s) \cdot B_j(s)) \, \mathrm{d}s$ multiplied by the target amplitude vector b. On the right we have the original basis amplitudes vector a multiplied by a projection matrix P, where:

$$P_{ij} = \int_S (B_i(s) \cdot \int_S (C(\omega, s) A_j(\omega)) \, \mathrm{d}\omega) \, \mathrm{d}s \tag{7.20}$$

The final projected values after convolution with $C(\omega_i, \omega_o)$ are given by:

$$b = G^{-1} \times P \times a \tag{7.21}$$

If we instead have a vector of raw moments m which have been projected against the basis functions of A, this becomes:

$$b = (G_B^{-1} \times P \times G_A^{-1}) \times m \tag{7.22}$$

This can be folded into a single matrix multiplication, where the matrix, $(G_B^{-1} \times P \times G_A^{-1})$, is pre-computed offline.

In practice, P can be generated through Monte Carlo integration. In a primary loop, a number of pseudo-random sample directions on the unit sphere are generated, and $B_i(s)$ is evaluated for each i. Then, for each primary sample direction s, the value of $I_j = \int_S (C(\omega, s) A_j(\omega)) d\omega$ is computed for each j through Monte Carlo integration; in most cases this can be done by importance sampling $C(\omega, s)$ for ω. The product of B_i and I_j for each i and j can then be added, averaging over all primary sample directions to produce the final projection matrix P.

If the integration domain is defined to be over the hemisphere rather than the sphere, the integration of both the basis function and the BRDF must be defined over that hemisphere. In practice, this means that any directions importance sampled from the BRDF that point outside the hemisphere must have a

weight of zero; this is equivalent to uniformly sampling the hemisphere and weighting by the BRDF but converges more quickly.

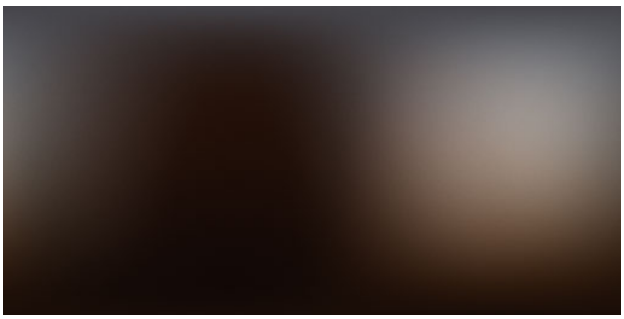
Note that for specular BRDFs such as GGX $C(\omega, s)$ can be formed by fixing the roughness α and either the view or normal direction (for example, to parameterise the encoded function by the view direction around a fixed normal of $(0, 0, 1)$, or to instead isotropically parameterise by assuming the view direction is always aligned with the normal).

The choice of source basis A determines how accurately the encoded signal in B reconstructs the original signal. Put another way, if b is intended to represent the convolution of the original signal $f(s)$ with some BRDF $C(\omega, s)$, the accuracy of this method depends on how well $\sum_i a_i A_i(s)$ represents the original radiance signal. For Ambient Dice, for example, the quality loss of using the same basis to store the intermediate radiance is minimal (at most around 15% increased RMSE for diffuse) since BRDFs blur the result; a comparison is shown in (c) and (d) of Figure 7.9.

Spherical harmonics (Section 8.1) are an orthonormal basis function. In the case of orthornormal basis functions, the Gram matrix G and its inverse G^{-1} are both the identity matrix, and the projection matrix P is a diagonal matrix. This is why spherical harmonics can be efficiently convolved with the Lambertian diffuse BRDF: multiplication of the basis coefficients b by $G_B^{-1} \times P$ requires only multiplication along the non-zero diagonal.

An alternative approach to using the projection matrix, suitable for use with the progressive least--squares encoding technique, is to project each radiance signal into the target space during the solve. For each sample s with direction ω_s, a random BRDF importance sampled direction ω_r on the hemisphere of ω_s is chosen. The radiance signal is then multiplied by the BRDF, given a normal of ω_r and a light direction of ω_s, and ω_r is used as the direction in which to evaluate the basis functions. This technique is necessarily noisier than a proper projection since we are discarding much of the information from each radiance signal – namely, the product of that signal with the BRDF evaluated with every other possible normal direction – but the cost is lower than the full matrix multiplication and non-negative solves are supported.

Figure 7.9: Comparison of Ambient Dice SRBF (Section 8.3) Lambertian irradiance representations on the Wells HDR environment map [85]



(a) Monte Carlo importance sampled irradiance

(b) Diffuse polynomial fit (Section 8.14) to a radiance Ambient Dice (twelve-lobe support) [RMSE: 0.00735]

(c) Irradiance-encoding Ambient Dice generated by directly solving for the ground truth irradiance (six-lobe support) [RMSE: 0.00964]

(d) Irradiance-encoding Ambient Dice generated by projecting from a radiance-encoding Ambient Dice SRBF (six-lobe support, Section 7.4) [RMSE: 0.00976]

Chapter 8

Families of Spherical Basis Functions

Given a method for encoding spherical basis functions (Chapter 7), the next decision to be made is which spherical basis function to encode into. An overview of prior work is given in Section 2.2. Rather than trying to comprehensively evaluate those methods in comparison with each other, this chapter will focus in on two particular families of basis function: spherical Gaussians [9] and Ambient Dice [12]. These two families are of particular interest due to their ability to reasonably accurately approximate indirect specular highlights, providing high visual quality at the cost of increased storage requirements and (in the case of spherical Gaussians) increased computational overhead. In addition, a brief overview of spherical harmonics is given due to their ubiquity and their unique property of orthonormality in the context of Section 7.4.

It should be noted that the ability for Ambient Dice to represent indirect specular is a novel and notable contribution of this work. In addition, the method by which Ambient Dice can represent indirect specular may be useful and applicable to a wide range of basis functions in future work.

8.1 Spherical Harmonics

Spherical harmonics (introduced in the context of lighting by Ramamoorthi and Hanrahan in 2001 [27] and used fairly ubiquitously since then) are a family of basis functions that have the useful property that all of its functions are orthonormal over a sphere.[1] Orthonormality means that for any two basis

[1] Technically, the family of spherical harmonics that are orthonormal are known as Laplace's spherical harmonics; in general, when we refer to spherical harmonics in the context of lighting we mean Laplace's spherical harmonics.

functions $B_i(s)$ and $B_j(s)$ in this family:

$$\int_S B_i(s)B_j(s)\,\mathrm{d}s = \begin{cases} 1 & i = j \\ 0 & \textit{otherwise} \end{cases} \tag{8.1}$$

In following with the definition of the Gram matrix from Section 7.2, note that this means the Gram matrix for any combination of spherical harmonics is the identity matrix. For this reason, performing least-squares encoding with spherical harmonic basis functions amounts to simply projecting the function values onto the basis functions. This orthonormality also has other useful properties; for example, convolution with the cosine function over the hemisphere can be done as a simple dot product (i.e. element-wise multiplying the basis coefficient vector b by the cosine convolution vector c), as discussed in Section 7.4.

Spherical harmonics are separated into *bands*, where each successive band contains higher-frequency information and requires a greater number of coefficients. The first band is simply a constant term, representing the average value over the entire sphere; the second band's basis functions are defined to represent the three components of a 3D direction vector, where one coefficient is required per component.

Due to the increasing number of coefficients required to represent high-frequency detail, spherical harmonics are usually used exclusively to represent diffuse lighting, which the first three bands (i.e. the first nine basis functions) represent very accurately [27]. In order to represent higher-frequency functions such as specular response, however, other families of basis functions are more useful.

In addition, since the basis functions for spherical harmonics differ with each band, the evaluation of each basis function (including for BRDF convolution) is different for each band. By contrast, bases comprised of radial basis functions in varying directions can be evaluated in the same manner for every basis component function, including for BRDF evaluation.

8.2 Spherical Gaussians

Spherical Gaussians (SGs) are spherical functions that can be used as part of a linear basis. Introduced by Wang et. al. in 2009 [9], spherical Gaussians were first adapted as an alternative to spherical harmonics in irradiance volumes and lightmaps by Pettineo and Neubelt, who presented their work in *Advanced Lighting R&D at Ready At Dawn Studios* at SIGGRAPH 2015 [10]. They are parameterised by three variables: μ, the amplitude; the lobe axis vector $\vec{p}$; and the lobe sharpness, λ. Radiance over a spherical or

hemispherical domain is often defined as the sum of a set of spherical Gaussian lobes, with the evaluation of a lobe in a particular direction defined as:

$$G(\vec{\omega}; \vec{p}, \lambda, \mu) = \mu e^{\lambda(\vec{\omega} \cdot \vec{p} - 1)} \tag{8.2}$$

The core advantage of spherical Gaussians over spherical harmonics is their ability to represent relatively high-frequency detail in fewer coefficients than spherical harmonics would necessitate. This enables spherical Gaussians to be used for representing indirect specular illumination, yielding plausible results for rough surfaces with wide specular lobes.

Spherical Gaussians are isotropic, meaning that they are a poor fit for anisotropic specular lobes such as those generated by the commonly-used Trowbridge-Reitz (GGX) distribution [89]. In 2013, Xu et. al. introduced Anisotropic Spherical Gaussians (ASGs) [90] which can much more closely fit anisotropic specular lobes. ASGs can be convolved with spherical Gaussian light sources, enabling lighting to be stored as a sum of SG lobes while still being evaluated for an anisotropic specular lobe.

In general, spherical Gaussians are stored as a set of lobes representing radiance, which can then be used to reconstruct either diffuse irradiance or specular. In my implementation, ASGs are used for specular reconstruction as per Pettineo and Neubelt, and for diffuse reconstruction I make use of Stephen Hill's irradiance curve fit [91].

Following Pettineo and Neubelt, I form a linear basis of spherical Gaussians by fixing the lobe directions evenly around a sphere or hemisphere and experimentally selecting a lobe sharpness. Although this biases the directionality of light sources towards the lobe directions, it does mean that only one coefficient per colour channel needs to be stored for each lobe, whereas including lobe directions and sharpnesses would double the storage and bandwidth costs.

8.3 Ambient Dice

Ambient Dice are a recent representation for signals on a unit sphere. Introduced by Iwanicki and Sloan in 2017 [12], Ambient Dice can represent irradiance signals with an accuracy between L3 and L4 spherical harmonics at substantially lower bandwidth cost.

In *Ambient Dice*, Iwanicki and Sloan presented two main separate variants of the basis using different reconstruction methods. The first, hybrid Bézier patch reconstruction, consists of locally-supported basis

functions and has a high ALU[2], bandwidth, and storage cost, but excellent quality; the second method uses a mixture of $\cos^2$ and $\cos^4$ lobes, with significantly lower ALU and bandwidth costs but also lower quality, particularly on high-frequency signals.[3] As a third alternative, they also propose encoding only luminance using the first method and using spherical linear interpolation for the chrominance, resulting in lower bandwidth cost than either representation but ALU cost near-identical to the Bézier patch.

These reconstruction methods are applied to the twelve vertices of a regular icosahedron (e.g. a twenty-sided die). For the Bézier patch variant, every direction lies on one of the twenty triangles formed by these vertices, and only the three vertices that comprise each triangle need to have data fetched to reconstruct the value. For the cosine-lobe variant, the six vertices lying on the same hemisphere as the direction being queried all need to be fetched. Offsetting that cost is the fact that the cosine-lobe variant requires only a single value per colour channel to be stored at each vertex, whereas the Bézier patch needs three: the value and two directional derivatives in perpendicular directions.

The cosine-lobe basis functions are defined as:

$$\text{CosineLobe}(x) = 0.7 \times \frac{1}{2}x^2 + 0.3 \times \frac{5}{6}x^4 \tag{8.3}$$

$$B_i(s) = \begin{cases} \text{CosineLobe}(\cos(\omega)), \cos(\omega) \geq 0 \\ 0, \qquad \text{otherwise} \end{cases} \tag{8.4}$$

where $\cos(\omega)$ is the dot product between the normalised icosahedron vertex direction v_i and the sample direction s. Note that this basis function, when used with lobes aligned with the vertices of an icosahedron, forms a partition of unity; at every point on the sphere the unscaled basis functions' values add up to one. In addition, each component of the basis function – $\frac{1}{2}x^2$ and $\frac{5}{6}x^4$ – also independently forms a partition of unity; therefore, this basis function is just one possible blend of the two components, and was chosen by Sloan and Iwanicki as providing the best appearance.

The hybrid Bézier patch's basis functions are more complex; for a full description, refer to Iwanicki and Sloan's original paper [12]. I also hope to contribute an implementation to the open source Probulator tool [87] in the near future.

[2] ALU refers to the number of instructions that use the arithmetic logic unit on the GPU and the cost of those instructions.

[3] This cosine-lobe basis function is a special case of axial moments, as introduced by Arvo in 1995 [92]. More recently, axial moments have seen use in approximating lighting due to clipped polygons (Belcour et. al. 2018) [93].

Unlike spherical Gaussians, the Ambient Dice family of basis functions was, in prior work, used to directly encode the signal to be reconstructed: rather than reconstructing irradiance from a set of radiance lobes (as is done with spherical Gaussians), irradiance is directly stored, and likewise for any other signal. This is particularly important for the Bézier patch variant, which relies on local support to be efficient; reconstructing a diffuse or rough specular signal implies fetching data from over a hemisphere and is prohibitively expensive in bandwidth.

Ambient Dice are defined over the sphere, making them seemingly less suitable for encoding hemispherical signals. However, it is worth noting that the locally-supported Bézier patch variant requires only nine vertices to be stored when representing signals defined on a hemisphere since the other three vertices will always have zero contribution.

In the remainder of this chapter, I will propose a number of new uses for the cosine lobe variant of Ambient Dice. In particular, I will provide methods for evaluating diffuse or specular irradiance directly from a radiance-encoding Ambient Dice, mirroring the functionality of spherical Gaussians at higher quality and lower computational cost.

8.3.1 Ambient Dice's Cosine-Lobe Basis on the Hemisphere

In addition to being distributed over the sphere, it is also possible to use the cosine-lobe basis functions in an arrangement targeting the hemisphere. For example, we can omit the three vertices that are unused in the Bézier configuration to yield a nine-vertex basis. While these vertices would have a non-zero contribution in the globally-supported cosine-lobe variant, their contribution is expected to be negligible since they primarily support the zero values on the negative hemisphere.

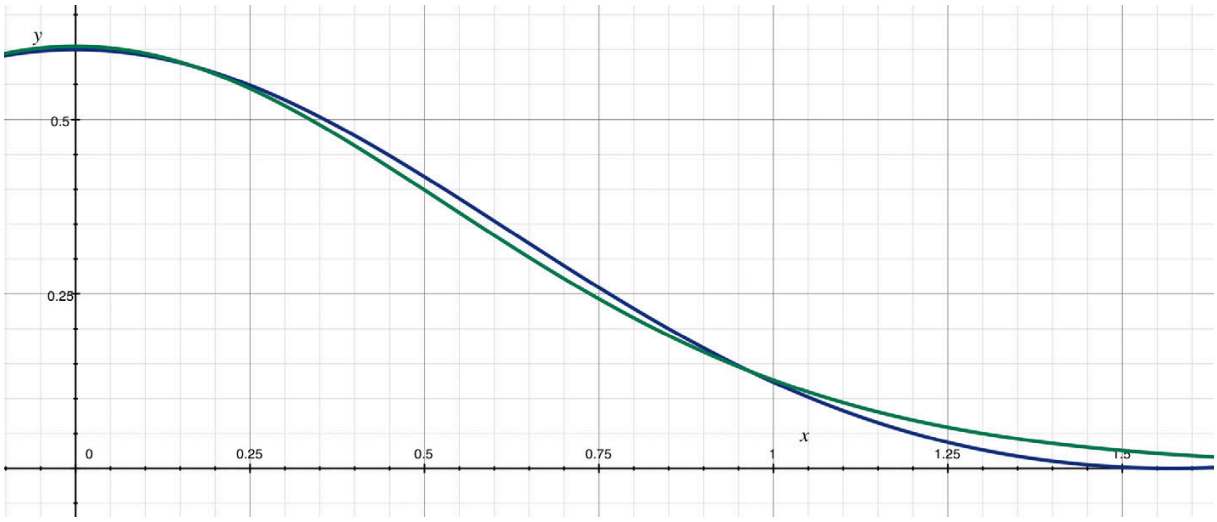
Using the $B_i(s)$ defined in Equation 8.3, we can define the basis vertices v to be:

$$\begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \\ v_7 \\ v_8 \\ v_9 \end{bmatrix} = \begin{bmatrix} 0.5257 & -0.3035 & 0.7947 \\ -0.5257 & -0.3035 & 0.7947 \\ 0.0 & 0.6071 & 0.7947 \\ 0.8507 & 0.4911 & 0.1876 \\ -0.8507 & 0.4911 & 0.1876 \\ 0.0 & -0.9822 & 0.1876 \\ -0.8507 & -0.4911 & -0.1876 \\ 0.8507 & -0.4911 & -0.1876 \\ 0.0 & 0.9822 & -0.1876 \end{bmatrix}$$

These vertices have been rotated from the original Ambient Dice configuration so that their average value is oriented towards $(0, 0, 1)$. Using this configuration rather than vertices chosen from the upper hemisphere of Vogel's sphere as is done in Probulator [87] reduces diffuse irradiance reconstruction error by a factor of around five on a range of environment maps.

8.3.2 Relationship to Spherical Gaussians

Figure 8.1: A spherical Gaussian lobe with $\lambda = 3.4$ (green) against the cosine-lobe variant of Ambient Dice (blue).



Note the longer tail of the spherical Gaussian lobe, which continues to be positive-valued over the entire sphere, whereas the cosine lobe has no influence at angles past $\frac{\pi}{2}$.

In prior work, the lobe directions for spherical Gaussians were chosen by sampling Vogel's sphere,

forming a spiral-like shape; then, the lobe sharpnesses were more or less arbitrary. However, it turns out that spherical Gaussians can form a reasonable approximation to the cosine-lobe variant of Ambient Dice, and that this approximation improves the quality of irradiance reconstruction from radiance-encoding SG lobes. On the 'pisa' HDR environment map [88], for example, the RMSE of spherical Gaussians using Vogel's sphere directions is 0.00828, whereas using icosahedron directions lowers it to 0.00576.

If, in addition to choosing the lobe directions, the lobe sharpness λ is chosen such that the shape of the Gaussian approximates that of the cosine-lobe variant of Ambient Dice, the irradiance reconstruction error can be lowered further, at the cost of diffused and less accurate specular. A value of around $\lambda = 3.4$ is a good fit, depending on which part of the curve you wish to fit most closely (Figure 8.1).

It is also possible to convert between Ambient Dice and spherical Gaussians with fixed lobes in the same way as between any other linear bases; the method in Section 7.4 for convolving with an arbitrary BRDF applies where the BRDF is a delta function such that:

$$C(\omega, s) = \begin{cases} 1 & \omega = s \\ 0 & \text{otherwise} \end{cases} \tag{8.5}$$

8.4 Evaluating BRDFs from Radiance Ambient Dice

Rather than converting a radiance-encoding Ambient Dice into one that represents the BRDF-weighted irradiance (Section 7.4), it is also possible to directly evaluate the radiance-encoding Ambient Dice for the irradiance. This is mostly useful when you want to store a single representation for radiance and evaluate multiple BRDFs from it.

Consider that the irradiance $I(\omega_o)$ is the integral over the hemisphere surrounding ω_o of the radiance multiplied by the BRDF $f_{br}(\omega_i, \omega_o)$. When using a linear basis to represent radiance, this becomes:

$$I(\omega_o) = \int_\Omega \sum_j b_j B_j(\omega_i) f_{br}(\omega_i)\,\mathrm{d}\omega_i \tag{8.6}$$

$$= \sum_j b_j \int_\Omega B_j(\omega_i) f_{br}(\omega_i)\,\mathrm{d}\omega_i \tag{8.7}$$

$$= \sum_j b_j C_j(\omega_o), \text{where } C_j(\omega_o) = \int_\Omega B_j(\omega_i) f_{br}(\omega_i)\,\mathrm{d}\omega_i \tag{8.8}$$

This does mean that the Ambient Dice must be evaluated with global support; whereas Ambient Dice generally have local support (the six lobes on the hemisphere for the cosine-lobe variant or the three vertices of each triangle for the Bézier patch variant), the BRDF collects radiance from over a hemisphere and therefore gathers irradiance either globally (for the cosine-lobe) or near-globally (for the Bézier patch). Due to the number of coefficients required for the Bézier patch, this approach is therefore only practical for the cosine-lobe variant.

The fits described in this section will, unless otherwise specified, assume that the basis functions are defined over the sphere. Defining the basis functions over the hemisphere requires that the fits consider not only the proximity of the sample direction to each lobe but also how much the basis-scaled BRDF lobe will be clipped by the hemisphere, which is more difficult to compute.

8.4.1 Diffuse Reconstruction from Cosine-Lobe Ambient Dice

If we restrict ourselves to the cosine-lobe variant of Ambient Dice and a Lambertian BRDF (Equation 8.3), the diffuse irradiance from each lobe is analytically evaluable. Given the angle θ_{lobe} between a given lobe v_i and the normal direction ω_o, the irradiance over the hemisphere Ω centred on ω_o is given by:

$$\begin{aligned} I_i(\omega_o) &= \int_\Omega B_i(s)\frac{\omega_o \cdot s}{\pi}\,\mathrm{d}s \\ &= \int_\Omega C(\max(\cos(\theta_{lobe})\frac{\omega_o \cdot s}{\pi}\,\mathrm{d}s \\ &= \int_\Omega (0.35 \times \max(\cos(\theta_{lobe}), 0)^2 + 0.25 \times \max(\cos(\theta_{lobe}), 0)^4)\frac{\omega_o \cdot s}{\pi}\,\mathrm{d}s \end{aligned}$$

This must be evaluated as a surface integral over the hemisphere Ω. The coordinate space will be

defined such that the Cartesian coordinates of a spherical pair (ϕ, θ) are given by:

$$\text{Cartesian}(\theta, \phi) = (\sin(\theta)\sin(\phi), \sin(\theta)\cos(\phi), \cos(\theta)) \tag{8.9}$$

Let the lobe direction in this coordinate space be determined by the angle with the normal θ_{lobe} and correspond to:

$$\textit{lobeDirection} = (\sin(\theta_{lobe}), 0, \cos(\theta_{lobe})) \tag{8.10}$$

Due to the clamped cosine, the integration bounds must be set carefully. Translated into a surface integral on the hemisphere we can see that there are two cases: one where θ_{lobe} is greater than $\frac{\pi}{2}$, in which case it only intersects the right half of the hemisphere, and one where θ_{lobe} is less than $\frac{\pi}{2}$, in which case the integral intersects both the left and right halves.

The first case is the most straightforward, where the irradiance I_i is given by:

$$\text{dot}(\theta, \phi, \theta_{lobe}) = \sin(\theta_{lobe})\sin(\theta)\sin(\phi) + \cos(\theta_{lobe})\cos(\phi)$$
$$I_i(\theta_{lobe}) = \int_0^{\pi}\int_{\theta_{lobe}-\frac{\pi}{2}}^{\frac{\pi}{2}} C(\text{dot}(\theta, \phi, \theta_{lobe}))\frac{\cos(\theta)}{\pi} d\theta d\phi \text{ , where } \theta_{lobe} > \frac{\pi}{2}$$

The definition for $\text{dot}(\theta, \phi, \theta_{lobe})$ falls out of the definition of the dot product in our coordinate space, and is equal to the cosine of the angle between (θ, ϕ) and the lobe direction.

With the help of MATLAB [94] or a similar symbolic computing tool, we can see that $I_i(\theta_{lobe})$ comes out to be:

$$
\begin{aligned}
I_i(\theta_{lobe}) = & \frac{1167\ \cos\left(6\,\theta_{lobe}\right)}{163840} - \frac{313\ \cos\left(4\,\theta_{lobe}\right)}{61440} - \frac{8959\ \cos\left(2\,\theta_{lobe}\right)}{245760} + \frac{5\ \cos\left(8\,\theta_{lobe}\right)}{49152} \\
& - \frac{35\ \cos\left(10\,\theta_{lobe}\right)}{98304} + \frac{67\ \sin\left(2\,\theta_{lobe}\right)}{960} + \frac{\sin\left(4\,\theta_{lobe}\right)}{384} + \frac{113\ \cos\left(2\,\theta_{lobe}\right)}{5760\,\pi} \\
& - \frac{\cos\left(4\,\theta_{lobe}\right)}{576\,\pi} - \frac{53\ \cos\left(6\,\theta_{lobe}\right)}{2560\,\pi} - \frac{\cos\left(8\,\theta_{lobe}\right)}{4608\,\pi} + \frac{5\ \cos\left(10\,\theta_{lobe}\right)}{4608\,\pi} \\
& + \frac{1}{512\,\pi} + \frac{67\,\mathrm{atan}\left(\frac{\sqrt{2}\left(\cos\left(\frac{\theta_{lobe}}{2}\right)-\sin\left(\frac{\theta_{lobe}}{2}\right)\right)}{2\,\cos\left(\frac{\theta_{lobe}}{2}-\frac{\pi}{4}\right)}\right)\ \sin\left(2\,\theta_{lobe}\right)}{240\,\pi} \\
& + \frac{\mathrm{atan}\left(\frac{\sqrt{2}\left(\cos\left(\frac{\theta_{lobe}}{2}\right)-\sin\left(\frac{\theta_{lobe}}{2}\right)\right)}{2\,\cos\left(\frac{\theta_{lobe}}{2}-\frac{\pi}{4}\right)}\right)\ \sin\left(4\,\theta_{lobe}\right)}{96\,\pi} + \frac{2841}{81920}\text{, where } \theta_{lobe} \geq \frac{\pi}{2}
\end{aligned}
\tag{8.11}
$$

When θ_{lobe} is less than $\frac{\pi}{2}$ and therefore intersects both halves of the hemisphere, the integral is made up of two components:

$$
\begin{aligned}
I_i(\theta_{lobe}) = & \int_0^{\pi}\int_0^{\frac{\pi}{2}} C(\mathrm{dot}(\theta,\phi,\theta_{lobe}))\frac{\cos(\theta)}{\pi}d\theta d\phi\text{, representing the right hemisphere} \\
& + \int_0^{\pi}\int_0^{\frac{\pi}{2}-\theta_{lobe}} C(\mathrm{dot}(-\theta,\phi,\theta_{lobe}))\frac{\cos(\theta)}{\pi}d\theta d\phi\text{, representing the left hemisphere} \\
& \text{where } \theta_{lobe} < \frac{\pi}{2}
\end{aligned}
\tag{8.12}
$$

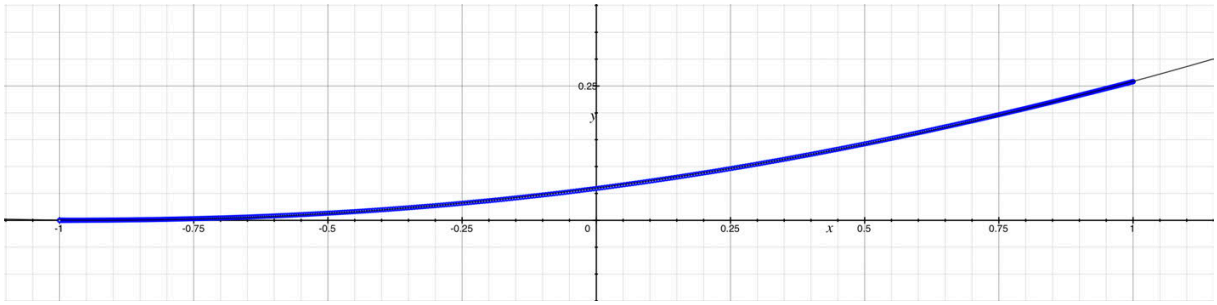
This evaluates to:

$$
\begin{aligned}
I_i(\theta_{lobe}) = & \frac{3\cos\left(\theta_{lobe}\right)^2}{40} + \frac{41\cos\left(\theta_{lobe}\right)^4}{192} + \frac{63\cos\left(\theta_{lobe}\right)^6}{320} - \frac{15\cos\left(\theta_{lobe}\right)^8}{32} \\
& + \frac{35\cos\left(\theta_{lobe}\right)^{10}}{192} + \frac{31\cos\left(\theta_{lobe}\right)^2}{120\,\pi} - \frac{23\cos\left(\theta_{lobe}\right)^4}{45\,\pi} - \frac{73\cos\left(\theta_{lobe}\right)^6}{120\,\pi} \\
& + \frac{17\cos\left(\theta_{lobe}\right)^8}{12\,\pi} - \frac{5\cos\left(\theta_{lobe}\right)^{10}}{9\,\pi} + \frac{31\,\theta_{lobe}\ \cos\left(\theta_{lobe}\right)\ \sin\left(\theta_{lobe}\right)}{120\,\pi} \\
& + \frac{\theta_{lobe}\cos\left(\theta_{lobe}\right)^3\ \sin\left(\theta_{lobe}\right)}{24\,\pi} + \frac{19}{320}\text{, where } \theta_{lobe} < \frac{\pi}{2}
\end{aligned}
\tag{8.13}
$$

Polynomial Approximation to the Integral

Instead of using this expensive analytic solution, we can also find a very close quadratic fit (shown in Figure 8.2) to $I_i(\theta_{lobe})$ which is parameterised by the angle between each vertex and the normal θ_{lobe}:

Figure 8.2: The imperceptibly-different quadratic curve fit $I_{i\,\text{approx}}(\theta_{lobe})$ (black) against the true value of $I_i(\theta_{lobe})$ (blue) for radiance-encoding cosine-lobe Ambient Dice, where the horizontal axis is $\cos(\theta)$.



$$I_{i\,\text{approx}}(\theta_{lobe}) = 0.05981 + 0.12918\cos(\theta_{lobe}) + 0.07056\cos^2(\theta_{lobe}) \tag{8.14}$$

The Lambertian irradiance in direction ω then becomes:

$$I(\omega_o) = \sum_i b_i I_i(\text{dot}(\omega_o, v_i)) \tag{8.15}$$

where v_i is the normalised direction and b_i is the radiance-encoded basis amplitude for the ith Ambient Dice vertex.

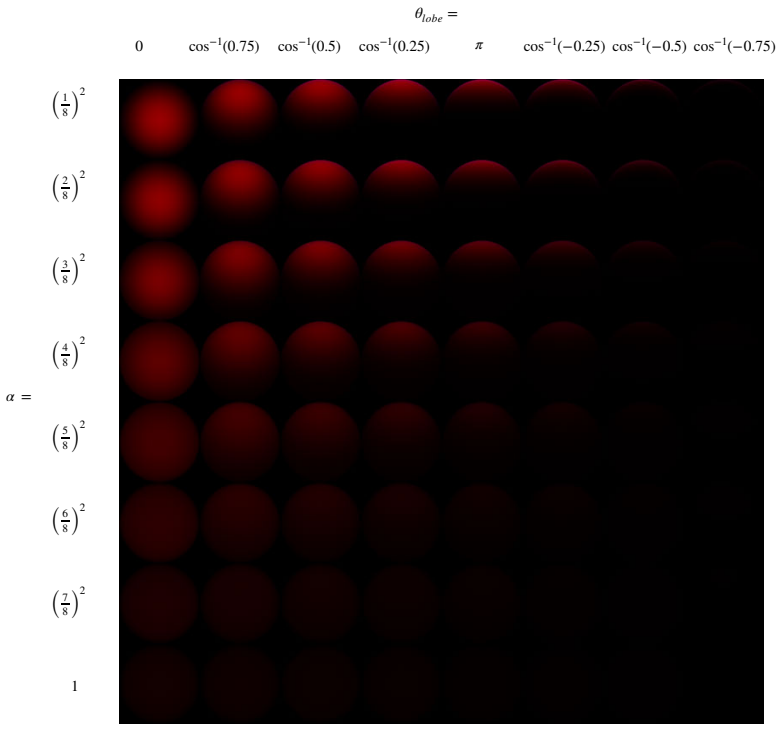
8.4.2 Specular Reconstruction from Cosine-Lobe Ambient Dice

In this section, a method for approximating the specular contribution from the cosine-lobe Ambient Dice basis function is provided. The method is simple to implement, requires little ALU, and needs only one additional texture lookup in addition to retrieving the per-lobe amplitudes. The error of the approximation over the parameter space is shown in Figures 8.3 and 8.4.

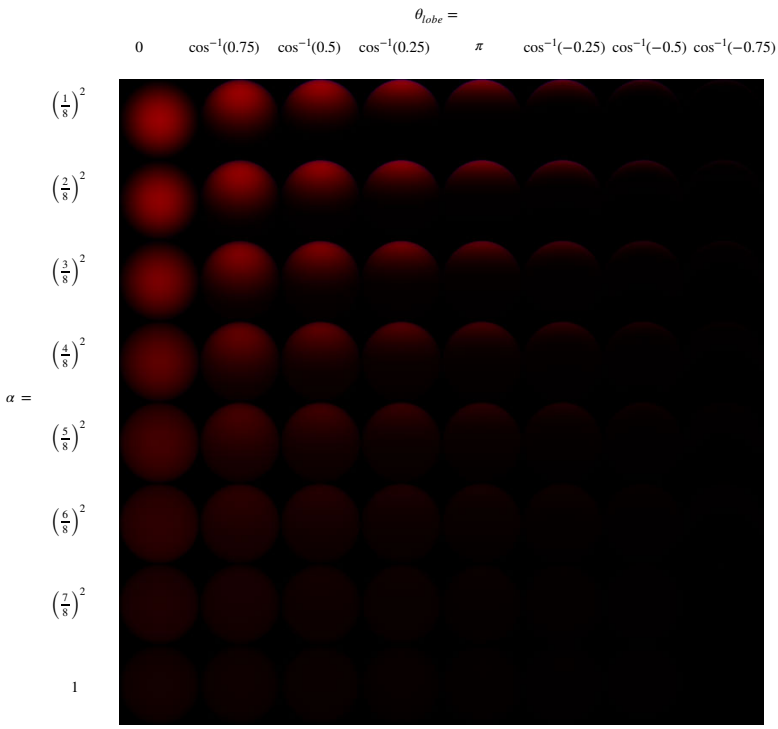
The approximation is made for the commonly-used single-scattering GGX specular model using the Smith height-correlated masking-shadowing function; see *Understanding the Masking-Shadowing Function in Microfacet-Based BRDFs* by Heitz (2014) for details [95]. However, the methodology may be applicable to a wide range of BRDFs and basis functions; in particular, an extension to multi-scattering GGX, while not provided here, is expected to be reasonably trivial.

In general, finding the integral of a specular BRDF with illumination from an arbitrary basis function is a non-trivial problem due to the large number of free parameters. For a general specular model parameterised by some isotropic roughness α, normal direction n, and reflectance at normal and grazing angles

Figure 8.3: Specular response from an Ambient Dice cosine lobe for single-scattering GGX.



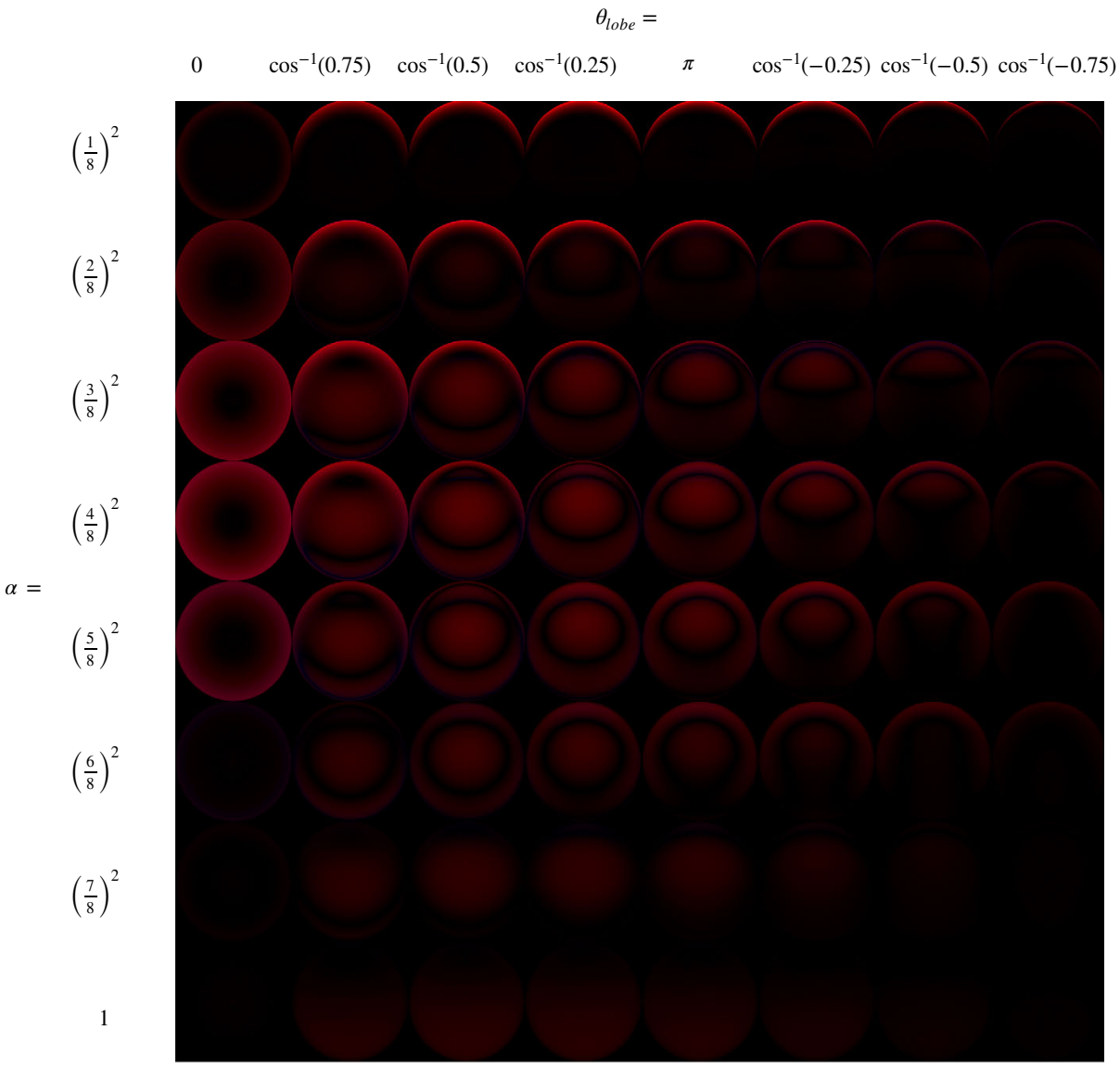
(a) Ground-truth response.



(b) Approximation with a 2D LUT and curve fit.

The f_0 scale is in red and the f_{90} scale is in blue. The position on each sphere indicates the viewing direction. RMSE of the approximation over the parameter space is 0.00749, with a maximum deviation from the true value of 0.0708.

Figure 8.4: Absolute difference between the ground truth and the 2D LUT approximation for the specular response from an Ambient Dice cosine lobe with single-scattering GGX, with exposure increased by 2 EV.



f_0 and f_{90}, the illumination from a light source in some linear basis is given by:

$$C_j(\omega_o) = \int_\Omega B_i(\omega_i) f_{br}(\alpha, \omega_i, \omega_o, n, f_0, f_{90}) \, \mathrm{d}\omega_i \tag{8.16}$$

The contribution of a linear combination of lobes in that basis $I_i(\omega_o)$ is a linear combination of C_j for each j (in other words, it is a weighted sum of each lobe's C_j) and is given by Equation 8.8.

Solving this integral analytically is challenging, and in many cases there may be no closed form solution. Instead, we are left with two options: for offline rendering, the integral may be evaluated using Monte Carlo integration, while for real-time, a fitted approximation or some look-up table should be used. Note that using Monte Carlo integration is overly expensive for real-time applications; visual artefacts are still readily apparent with as many as 32 samples when estimating $I_i(\omega_o)$.

Capturing the response of $I_i(\omega_o)$ in a fitted function is a difficult task in itself; there are a large number of input parameters which all affect the output in significant ways. However, there are two key observations that serve as a useful starting point:

- For a perfectly smooth specular reflector with α approaching zero, the BRDF becomes a delta function oriented in the surface's reflection direction (given the view direction). The radiance in this case is therefore simply the basis function evaluated in the reflection direction multiplied by the Fresnel response.

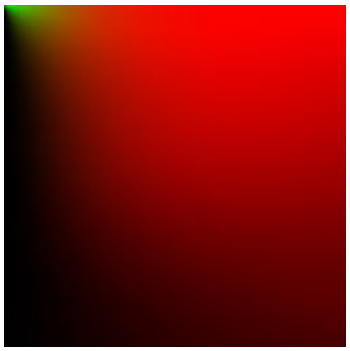
- For a very rough surface, the specular response will approach a Lambertian diffuse response. As such, scaling the diffuse model by the BRDF's response in a split-sum approximation is a reasonably close match to the ground truth:

  $C_j(\omega_o) \approx (\int_\Omega B_i(\omega_i) \, \mathrm{d}\omega_i)(\int_\Omega f_{br}(\alpha, \omega_i, \omega_o, n, f_0, f_{90}) \, \mathrm{d}\omega_i)$

In *Real Shading in Unreal Engine 4* (2013) [96], Karis showed that a specular BRDF's response can be precomputed into a 2D texture parameterised by the roughness value α and *NdotV*, where *NdotV* is the cosine of the angle between the surface normal and the viewing direction. This texture is known as the *DFG texture*, and is usually generated through Monte Carlo integration.

The DFG texture generally comprises of at least two channels, where for every value of α and *NdotV* one channel contains the scale for the material's f_0 and the other the scale for the material's f_{90} values. The DFG texture for specular Ambient Dice is shown in Figure 8.5.

Figure 8.5: The Ambient Dice DFG texture for single-scattering GGX. NdotV increases to the right and α increases downwards.



Using our earlier observation that the diffuse irradiance may be scaled by the specular BRDF response at high roughnesses, we can approximate the specular response for high roughnesses with:

$$I(\omega_o) = (f_0 \cdot DFG(\alpha, dot(n, \omega_o)).x + f_{90} \cdot DFG(\alpha, \text{dot}(n, \omega_o)).y) \cdot \sum_i b_i L(\omega_o) \tag{8.17}$$

where L is given by Equation 8.14, our polynomial approximation to the diffuse irradiance from an Ambient Dice cosine lobe.

We similarly multiply the reflectance at low roughnesses by the value stored in the DFG texture; rather than separately computing the Fresnel response, we can use the precomputed value.

Given a solution for each end of the roughness range, the next step is to find an approximate fit for roughness values in the middle of that range. The most immediately obvious solution is to linearly interpolate between the values for zero and maximum roughness based on some function $t(\alpha)$, where $t(\alpha)$ is a polynomial. In practice, a second function of α is also used to blend between the reflection direction and the normal direction as input for the diffuse fit. The coefficients for the polynomials can be found using a numerical fitting framework such as MPFIT [97] and are given in Listing 8.1:

Listing 8.1: Ambient Dice Specular Fit

```
float AmbientDiceCosineBasisFunction(float cosTheta) {
   float dotProduct = max(cosTheta, 0.f);
   float cos2 = dotProduct * dotProduct;
   float cos4 = cos2 * cos2;
   return 0.7f * (0.5f * cos2) + 0.3f * (5.f / 6.f * cos4);
}
```

```
const float kAmbientDiceParameters[8] = float[8](3.0498910522220495, -6.983002509990005,
    7.388270435580356, -2.662756921813306, -0.4005429486854629, 5.626699351644211,
    -6.040098716506305, 1.9006124935607012);

float EvaluateAmbientDiceLobeDiffuse(float cosTheta) {
   return 0.06 + 0.129 * cosTheta + 0.0697 * cosTheta * cosTheta;
}

float2 EvaluateAmbientDiceLobeSpecular(float3 lobeDirection, float3 viewDirection, float
    ggxAlpha, float2 lutValue) {
   float NdotLobe = dot(normal, lobeDirection);

   float3 reflectionDir = reflect(-viewDirection, normal);
   float RdotLobe = dot(reflectionDir, lobeDirection);

   float basisInMirrorDir = AmbientDiceCosineBasisFunction(RdotLobe);

   float sqrtAlpha = sqrt(ggxAlpha);

   float focusLerp = kAmbientDiceParameters[0] * sqrtAlpha + kAmbientDiceParameters[1] *
       ggxAlpha + kAmbientDiceParameters[2] * sqrtAlpha * ggxAlpha +
       kAmbientDiceParameters[3] * ggxAlpha * ggxAlpha;
   float diffuseParam = mix(RdotLobe, NdotLobe, saturate(focusLerp));
   float diffuse = EvaluateAmbientDiceLobeDiffuse(diffuseParam);

   float alphaLerp = kAmbientDiceParameters[4] * sqrtAlpha + kAmbientDiceParameters[5] *
       ggxAlpha + kAmbientDiceParameters[6] * sqrtAlpha * ggxAlpha +
       kAmbientDiceParameters[7] * ggxAlpha * ggxAlpha;

   float value = mix(basisInMirrorDir, diffuse, saturate(alphaLerp));
   return value * lutValue;
}
```

Applying this simple custom fit yields barely reasonable results; the general intensity values are approximately correct but the shape of the distribution is obviously incorrect. After some investigation, it became apparent that using the split-sum approximation throughout the entire range is the main source of error; post-multiplying the result of importance sampling the basis function according to the BRDF by the BRDF response yields high error and visual disparity.

Rather than using the split-sum approximation of the BRDF response with the basis function, we can instead build a separate look-up table based on our simple fit. Instead of containing the BRDF response, this table should hold the ground truth value for a particular angle between the surface normal and lobe direction divided by the value our simple approximation would yield. With this table, an exact solution can be evaluated for that particular view and lobe direction, with an approximation for all other parameter

values.[4]

When constructing the table, the integral is evaluated in tangent space, with the normal set to $N = (0, 0, 1)$. The view direction is given by $V = (0, \sin(\cos^{-1}(NdotV)), NdotV)$, and the lobe direction is some blend between the normal direction and the mirror reflection direction $R = (0, -\sin(\cos^{-1}(NdotV)), NdotV)$ depending on the roughness.

Listing 8.2 provides the implementation for generating the lookup texture. See *Sampling the GGX Distribution of Visible Normals* (Heitz 2018) [72] for definitions and/or explanations of the `sampleGGXVNDF` and `SmithGGXMaskingShadowingG2OverG1Reflection` functions.

Listing 8.2: Ambient Dice Lookup Texture Generation

```
// Sampling the GGX Distribution of Visible Normals
// Heitz 2018.
float SmithLambda(float cosThetaM, float alphaG) {
   float alphaG2 = alphaG * alphaG;

   float cosThetaM2 = cosThetaM * cosThetaM;
   float sinThetaM2 = 1.f - cosThetaM2;

   return 0.5f * (-1.f + sqrt(1.f + alphaG2 * sinThetaM2 / cosThetaM2));
}

// Sampling the GGX Distribution of Visible Normals
// Heitz 2018.
inline float SmithGGXMaskingShadowingG2OverG1Reflection(float3 incoming, float3 outgoing,
    float3 microfacetNormal, float alpha) {
   float VdotH = dot(incoming, microfacetNormal);
   float LdotH = dot(outgoing, microfacetNormal);

   float G1Inverse = 1.f + SmithLambda(incoming.z, alpha);

   float numerator = (VdotH > 0 ? 1.f : 0.f) * (LdotH > 0 ? 1.f : 0.f);
   float denominator = G1Inverse + SmithLambda(outgoing.z, alpha);
   return numerator / (denominator * G1Inverse);
}

float3 GGXDominantDirection(float3 N, float3 R, float roughness) {
   float smoothness = saturate(1.f - roughness);
   float lerpFactor = smoothness * (sqrt(smoothness) + roughness);
   return normalize(mix(N, R, lerpFactor));
}
```

[4] Note that *NdotV* and *NdotLobe* alone are insufficient to uniquely determine the integrated value; we also need to know the angle between the view direction and the lobe direction.

```
float2 IntegrateDFGAmbientDice(float NdotV, float ggxAlpha) {
   const uint sampleCount = 256u;
   const float sampleScale = 1.f / float(sampleCount);

   const float3 normal = float3(0, 0, 1);

   float3 viewDirection = float3(0, sqrt(1.f - NdotV * NdotV), NdotV);
   float3 R = reflect(-viewDirection, normal);
   float3 lobeDirection = GGXDominantDirection(normal, R, ggxAlpha);

   float fittedValue = EvaluateAmbientDiceLobeSpecular(lobeDirection, viewDirection,
       ggxAlpha, float2(1.f)).x;

   float2 groundTruth = float2(0.0); // for f0 and f90MinusF0
   for (uint sampleIt = 0u; sampleIt < sampleCount; sampleIt += 1u) {
      float2 sampleUV = hammersley2D(sampleIt, sampleCount);
      float3 H = sampleGGXVNDF(viewDirection, ggxAlpha, ggxAlpha, sampleUV.y, sampleUV.x);
      float3 lightDirectionTangent = reflect(-viewDirection, H);

      float Vis = SmithGGXMaskingShadowingG2OverG1Reflection(viewDirection,
          lightDirectionTangent, H, ggxAlpha);

      float f0Weight = 1.f;
      float f90MinusF0Weight = pow(1.f - saturate(dot(viewDirection, H)), 5.f);

      float basis = AmbientDiceCosineBasisFunction(dot(lobeDirection, lightDirectionTangent));

      float2 brdf = float2(f0Weight - f90MinusF0Weight, f90MinusF0Weight) * Vis;
      if (lightDirectionTangent.z > 0.f) {
         groundTruth += basis * brdf * sampleScale;
      }
   }

   return groundTruth / fittedValue;
}

kernel void GenerateAmbientDiceDFGTexture(texture2d<float, access::write> texture [[
    texture(0) ]],
                      ushort2 gid [[thread_position_in_grid]],
                      ushort2 gridSize [[threads_per_grid]]) {
   float2 uv = (float2(gid) + 0.5) / float2(gridSize);
   float NdotV = uv.x;
   float ggxAlpha = uv.y;

   float2 dfg = IntegrateDFGAmbientDice(NdotV, ggxAlpha);

   texture.write(float4(dfg, 0, 0), gid);
}
```

Table 8.1: Runtime overhead of indirect specular lightmaps.

No lightmaps	$1.39ms$
Ambient Dice (nine lobes)	$2.30ms$
Spherical Gaussians (nine lobes)	$3.27ms$

Times given are from GPU performance queries for rasterising the main view only. Forward shading was used, with front-to-back sorting to minimise overdraw.
Tests were performed on the Sponza Atrium [5] with a 2048×1862 lightmap. The lightmap was stored as four 32-bit floats per basis function; timings would likely be lower with a compressed format.

Results

In practice, this fit works very well, and produces results very close to the ground truth for high roughnesses. For a more accurate fit, a 3D lookup texture could be used, where the texture is indexed by the dot product between the normal and lobe directions in addition to *NdotV* and α.

I have not yet found a compensation factor for the portion of the BRDF that is clipped when the basis functions are defined on the hemisphere; however, this is a less significant issue for specular than for diffuse due to the narrower nature of low to medium roughness specular lobes.

Due to the inherent blurring in the Ambient Dice cosine-lobe basis, the specular reconstruction is only accurate compared with the ground truth for moderate to high roughness values. This is not a limitation of the specular reconstruction technique but rather of the limited bandwidth allotted to the basis functions, preventing the storing of high-frequency data.

Compared with spherical Gaussians, this fit produces results that more accurately match the ground truth. The Ambient Dice cosine-lobe basis produces more diffused highlights; however, as a consequence, it does not suffer from the point-light effect that spherical Gaussians suffer when configured with the high λ values necessary for those tight highlights.[5] Performance is also significantly better than for spherical Gaussians due to the much lower ALU overhead, as can be seen in Table 8.1.

A comparison of the Ambient Dice specular reconstruction compared to spherical Gaussians and the ground truth is given in Figure 8.6; note in particular the circular banners, the hanging vases, and the dark area in the back of the hall. Additionally, a detailed image comparison of reconstruction from Ambient Dice compared to spherical Gaussians is provided in Appendix C.

[5] High-λ spherical Gaussian lobes produce highly localised highlights, meaning that specular reconstruction from spherical Gaussian lobes often gives the appearance of point light sources rather than a continuous radiance field; Figure 7.4 shows an example of this.

8.4.3 Potential for Widespread Use

Radiance-encoding cosine-lobe Ambient Dice have the potential to be a new standard for storing high--quality indirect specular in real-time applications. They are inexpensive to evaluate, produce diffuse reconstruction that is at worst slightly higher error than L2 spherical harmonics and at best approaches L4 spherical harmonics, and produce specular reconstruction that is often of higher quality and lower error than spherical Gaussians at a significantly lower runtime cost.

When used in lightmap baking, cosine-lobe radiance-encoding Ambient Dice are a higher-quality but higher-cost alternative to L1 spherical harmonics, with the added bonus of providing indirect specular. However, they also have applicability in place of the localised cubemaps or baked irradiance volumes in widespread use today. Although the Ambient Dice require a high number of coefficients (twelve for spherical encoding compared to the nine required by L2 spherical harmonics), the same coefficients are used for both diffuse and specular. For surfaces with moderate to high roughness values, using cosine-lobe Ambient Dice removes the need for a cubemap texture fetch to retrieve the preconvolved specular irradiance.

Localised cubemaps or screen-space reflections will likely continue to be necessary for high-frequency specular reconstruction with low roughness values. Screen-space reflections using baked indirect lighting from cosine-lobe Ambient Dice as a fallback could potentially be a popular option, providing high visual quality at moderate cost. A comparison of indirect lighting techniques, including the combination of screen-space reflections with lightmaps, is given in Figure 8.7.

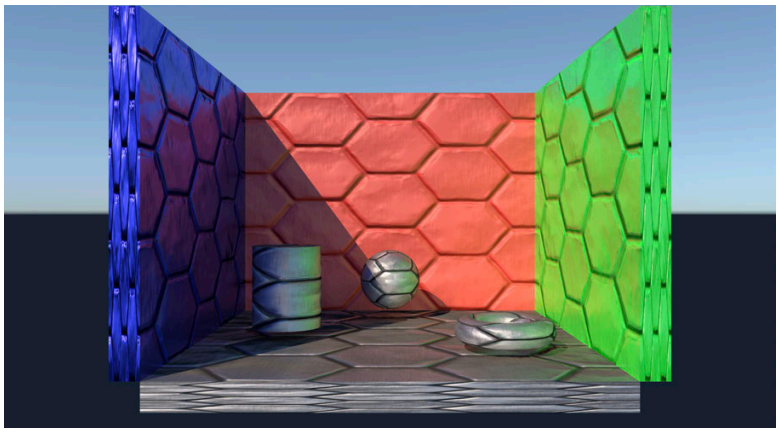
Figure 8.6: Indirect lighting from baked lightmaps in Sponza Atrium [5] with only single-scattering GGX materials.

(a) Path-traced reference

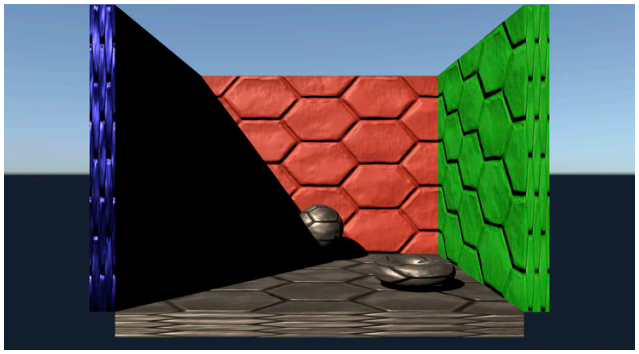
(b) Non-negative spherical Gaussians (12 lobes, $\lambda = 8$)

(c) Non-negative hemispherical cosine-lobe Ambient Dice (9 lobes)

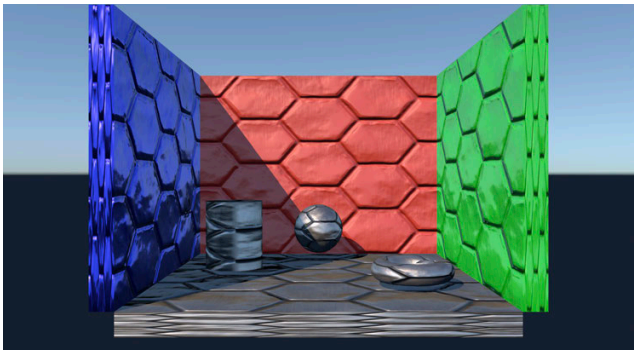
Figure 8.7: Comparison of various real-time indirect lighting techniques. Scene credit The Baking Lab [18].



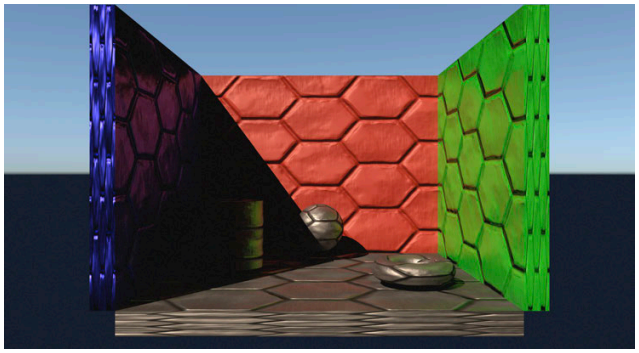
(a) Path traced reference



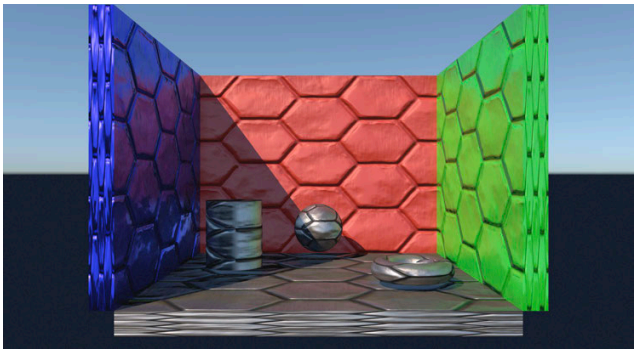
(b) Direct lighting



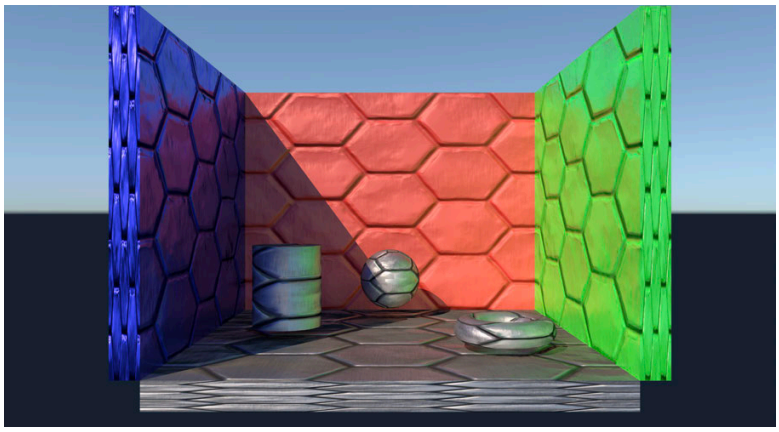
(c) Skybox probe (cubemap specular and L2 SH diffuse)



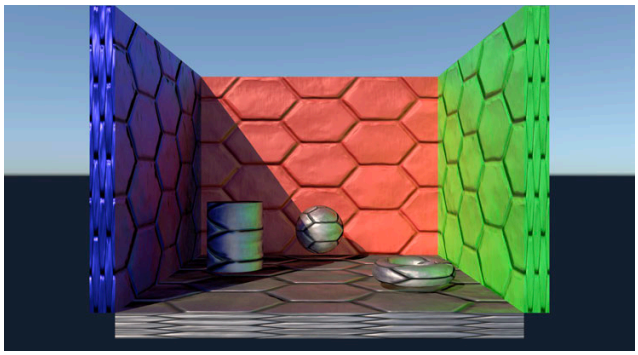
(d) Screen-space reflections



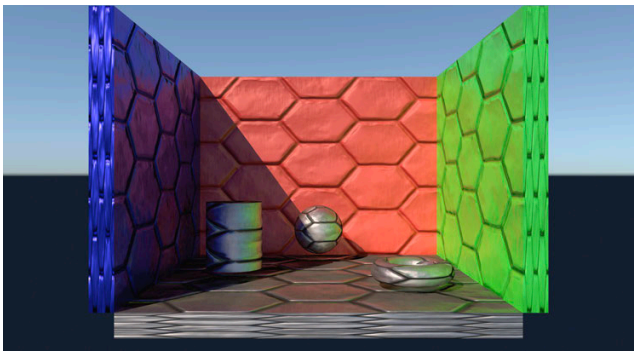
(e) Skybox probe and screen-space reflections



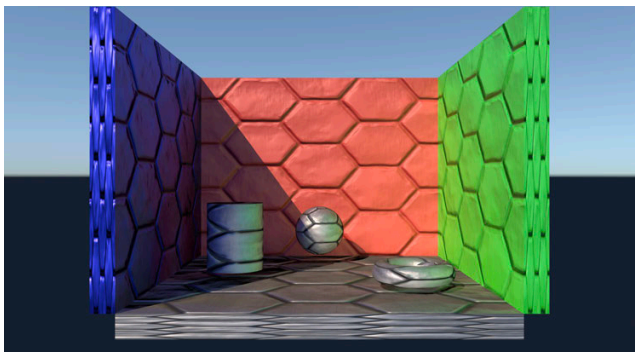
(a) Path traced reference



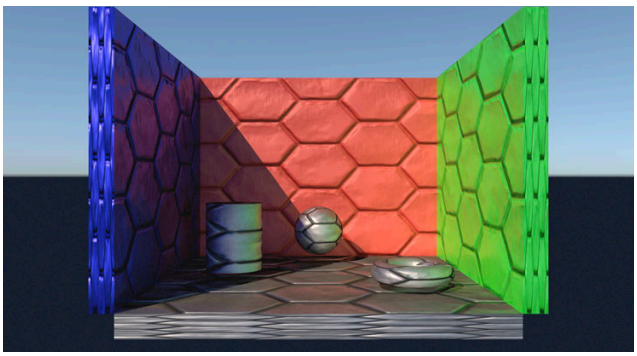
(f) Indirect lightmap with 12 spherical Gaussian lobes



(g) Indirect lightmap with 12 spherical Gaussian lobes and screen-space reflections

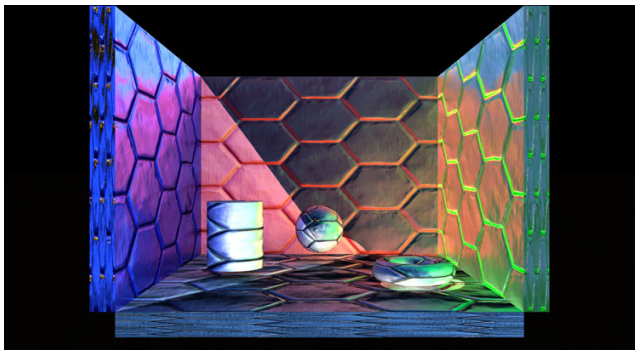


(h) Indirect lightmap with 9 Ambient Dice cosine-variant lobes

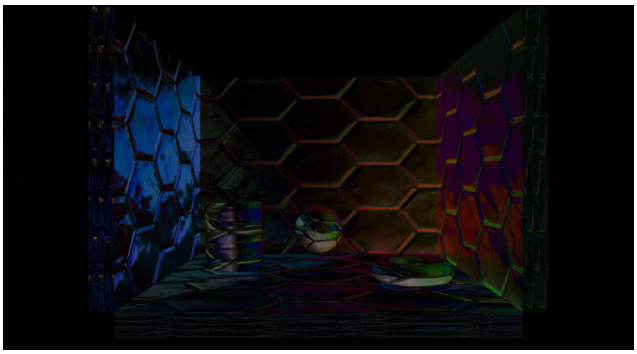


(i) Indirect lightmap with 9 Ambient Dice cosine-variant lobes and screen-space reflections

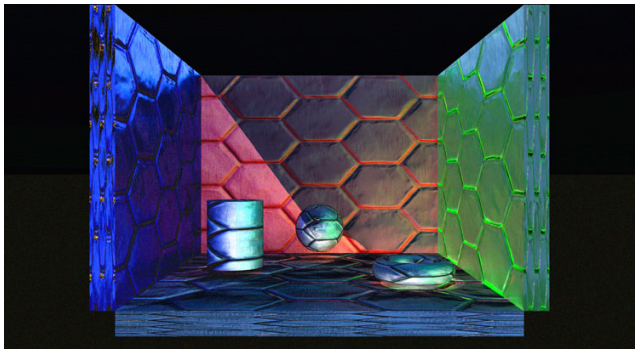
Figure 8.8: Comparison of various real-time indirect lighting techniques: absolute difference from path-traced reference. Intensities have been increased by 3 EV. Scene credit The Baking Lab [18].



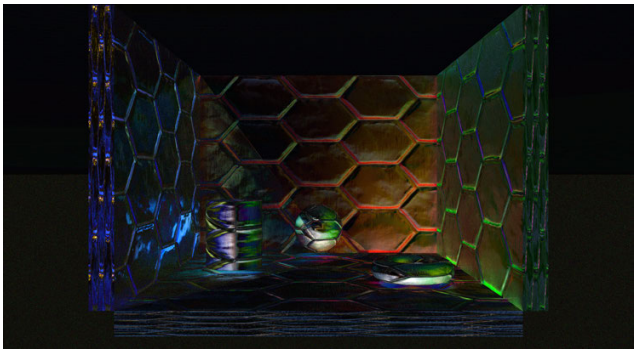
(a) Direct lighting



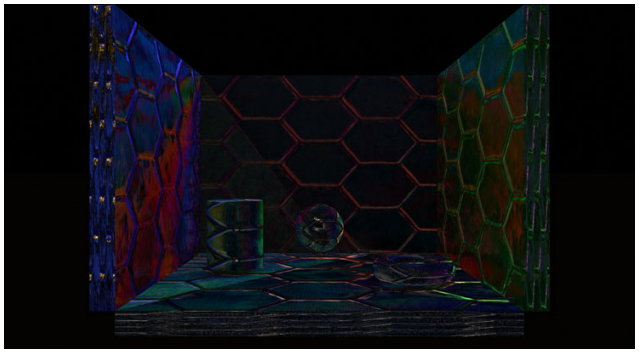
(b) Skybox probe (cubemap specular and L2 SH diffuse)



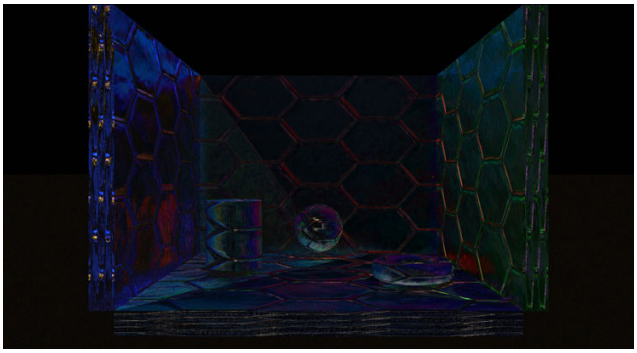
(c) Screen-space reflections



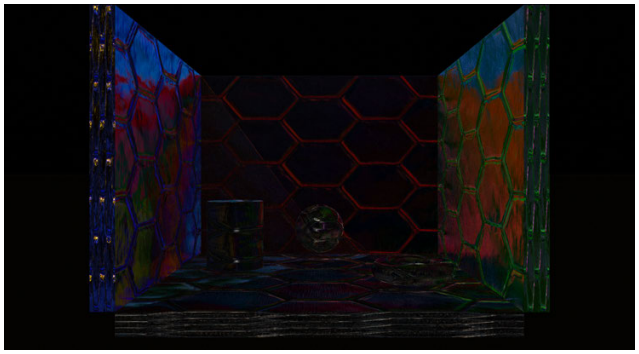
(d) Skybox probe and screen-space reflections



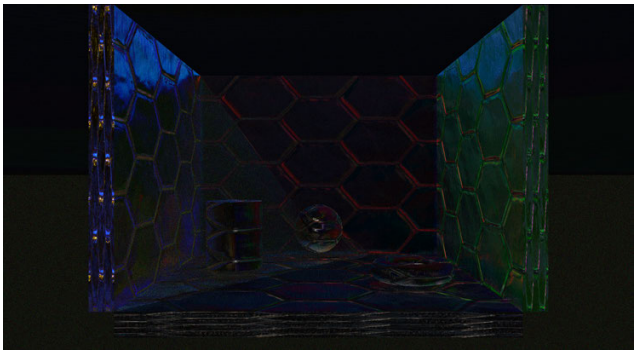
(e) Indirect lightmap with 12 spherical Gaussian lobes



(f) Indirect lightmap with 12 spherical Gaussian lobes and screen-space reflections



(g) Indirect lightmap with 9 Ambient Dice cosine-variant lobes



(h) Indirect lightmap with 9 Ambient Dice cosine-variant lobes and screen-space reflections

Chapter 9

Conclusion

This thesis has presented an overview of and a number of extensions to the state of the art in GPU path tracing and interactive lightmap generation, enabling artist-friendly production of high-quality global illumination in real-time applications. The stated goals of the thesis were to present a framework for building a GPU path-tracing lightmapper, to provide a good user experience by making that lightmapper performant, and to improve the visual quality of the produced lightmaps; the structures, techniques, and algorithms presented in this thesis have contributed to each one of these goals, and in combination form a complete system for path-traced lightmap generation on the GPU.

To recap some of its most notable contributions, this thesis has:

- Presented a detailed and optimised method for packing parameterised meshes into an atlas and generating samples from that parameterisation on the GPU.
- Described a range of considerations for GPU-based path tracing, including performance-quality tradeoffs enabled by integration with an existing rasteriser such as irradiance caching.
- Introduced a gather-based parallel method of sample accumulation that supports filtering, allows multiple samples to be taken per pixel in a single batch, and supports multi-operation modifications. In particular, this enables the progressive least-squares encoding method.
- Presented a novel method for progressive least-squares encoding of spherical basis functions, including support for approximate non-negative solves.
- Introduced methods to reconstruct diffuse and specular irradiance from radiance-encoding Ambient Dice using the cosine-lobe variant basis function.

GPU path-traced lightmaps present an exciting future for the content creation pipeline, enabling fast iteration and accessible, accurate global illumination. The democratisation that GPU-generated lightmaps imply for small and independent content creators – wherein the process can run interactively on an artist's hardware without requiring an overnight baking process or separate hardware – is particularly compelling.

As part of this thesis, an efficient implementation has been built that enables interactive lightmap baking for real-time applications. The hope is that the steps and processes detailed here will be useful for others wishing to do the same, and that those implementations, in turn, will help to enable the proliferation of incredible, well-illuminated experiences.

9.1 Limitations and Future Work

This thesis' system does not include a denoising filter, which would improve the apparent convergence rate for artists. The wavelet À-trous filter (Dammertz et. al., 2010) [98] would be a simple and effective addition and has already seen use in the Frostbite engine's progressive lightmap baker [4].

Modifications to the lighting conditions or object positions within a scene currently necessitate restarting the lightmap baking process, while modifications to object sizes and object addition require re-parameterisation of the lightmap. For localised changes, an adaptive method could be used to only regenerate the parts of the lightmap that the scene changes significantly affect; for example, the value of the new lightmap after a few samples could be compared with the value in the old lightmap, with the samples merged if the averages are sufficiently close.

Additionally, new, more robust methods to parameterise meshes would remove the remaining major hurdle in using lightmaps in content creation. Mesh parameterisation, even when automated, can easily fail and require manual correction if the mesh is non-manifold or has other issues. Techniques such as surfels (Barre-Brisebois et. al.) [1], wherein the scene surfaces are parameterised in a non-geometry-aware manner, may present a better workflow for artists.

More attention should be spent on optimising the surface shading kernels for a GPU path tracer. The high register usage in these kernels yields low occupancy (few threadgroups in flight) on the GPU hardware, and only moderate effort was made in minimising divergence within the surface shader. It is likely that highly optimised shading kernels could greatly outperform the implementation used here.

Building on camera-based lightmap sampling, a useful extension to the interactive GPU path tracer

may be to focus samples on particular objects within the scene. This could be implemented as a fairly simple extension of the current adaptive sampling mechanisms by restricting the sample domain to be only the lightmap UVs contained within the objects of interest.

Camera-based lightmap sampling also performs useful culling of lightmap geometry that is invisible from user perspectives. After an artist has fully inspected a scene using camera-based sampling, only the non-zero texels need to have further samples taken for the final bake process since those texels with zero samples are invisible to the user. This could also be used to re-parameterise the lightmap to remove invisible geometry, improving utilisation.

Irradiance caching is currently implemented using a simple Lambert-shaded direct illumination lightmap. As an extension, the approach proposed by Apers et al. [1] could be used wherein the converged lightmap texels are used to provide cached indirect illumination for future path tracing. If these converged lightmap texels were stored as a spherical basis function, then indirect radiance reconstruction from the irradiance cache would be possible; particularly interesting is the prospect of a hybrid approach, wherein the cached indirect radiance is used for rough materials while ray tracing is used for highly glossy materials, resulting in an accurate final result at reduced computational cost.

The approach for approximating specular from the Ambient Dice cosine-lobe basis function could see use in approximations for other basis functions, as well as for approximating multi-scattering specular. It could also be interesting to layer higher-frequency representations; for example, Iwanicki and Sloan suggest a subdivided icosahedron for a variant of Ambient Dice that can store higher-frequency illumination.

Spherical basis functions whose error is minimised over the hemisphere yield significantly improved images, yet result in poor BRDF reconstruction when the BRDF hemisphere (centred around the query direction) does not align with the integration hemisphere (which is fixed to the geometric surface normal). Finding better approximations for BRDF integrals over hemispherical basis functions would yield a significant quality improvement at no extra bandwidth cost.

Appendices

Appendix A

Validation and PBRT Compatibility

When testing the path tracer, it was useful to have some ground truth reference to validate its output. For my reference, I chose the PBRT renderer [13] due to the availability of multiple high-quality scenes within its file format and the high-quality documentation available.

I built a parser that can load scene files from the PBRT scene format into LlamaEngine (Appendix B.1). A best-effort attempt is made at preserving materials; LlamaEngine supports only a subset of the material models implemented within PBRT, and in particular has no support for procedural textures or arbitrary texture manipulations. In addition, the layered material models fundamentally differ between PBRT and LlamaEngine; in PBRT's `substrate` material the transmittance from the specular layer through to the diffuse layer depends upon the Fresnel reflectance at the incident angle, for example, whereas LlamaEngine uses a purely additive model.[1] Given these limitations, LlamaEngine cannot precisely match PBRT's output; however, in most cases the rendered image is very similar; see Figure A.1.

PBRT scenes present a number of challenges when being rendered within a rasteriser. The vertex normals are frequently inverted, for example, and the coordinate system freely shifts between being left--handed and right-handed, with face winding varying similarly. Addressing this was a matter of implementing detection for mismatching face winding (ensuring that the shading normal is oriented with the geometric normal, and flipping the winding order if it is not) and manually tweaking scenes file to mitigate the remaining issues.

In addition, the PBRT scene files are generally sadly unsuitable for lightmapping due to their high geometric complexity. Parameterising the PBRT scenes can take a number of hours and the resulting

[1] Note that neither model is physically correct, as is described in Section 6.3.1.

parameterisations have numerous issues that render the final result unusable. Additionally, many of the meshes in PBRT scenes are non-manifold and cannot be parameterised.

The path tracer must also be independently validated against the rasteriser since lighting must be consistent between both. In cases where a rasteriser may accurately represent a scene (i.e. with only punctual and directional lights and simple materials) a single bounce of the path tracer is a near-exact match to the rasteriser, with the only differences being in shadow handling; see Figure A.2.

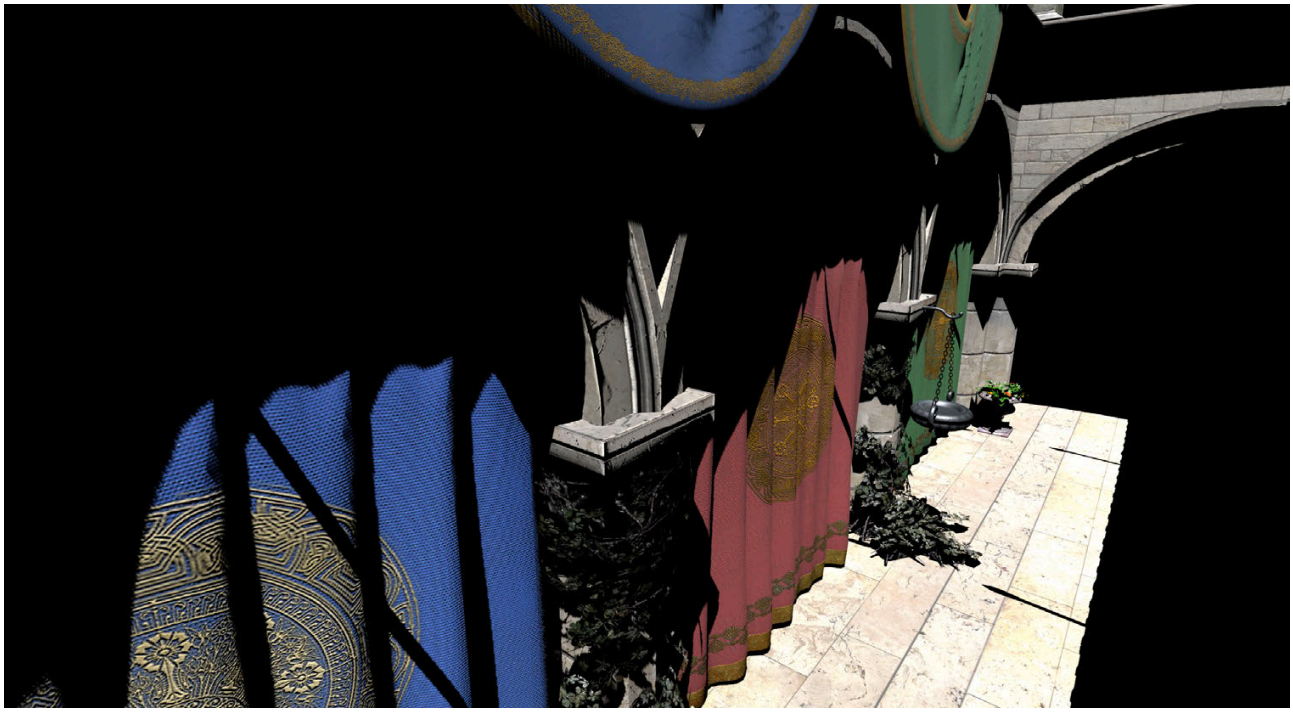
Figure A.1: 'bathroom' in PBRT and in LlamaEngine. Scene credit 'nacimus' [40].

(a) LlamaEngine GPU render. 4000 samples per pixel at 2400×1520 resolution. 550.9 seconds of render time.

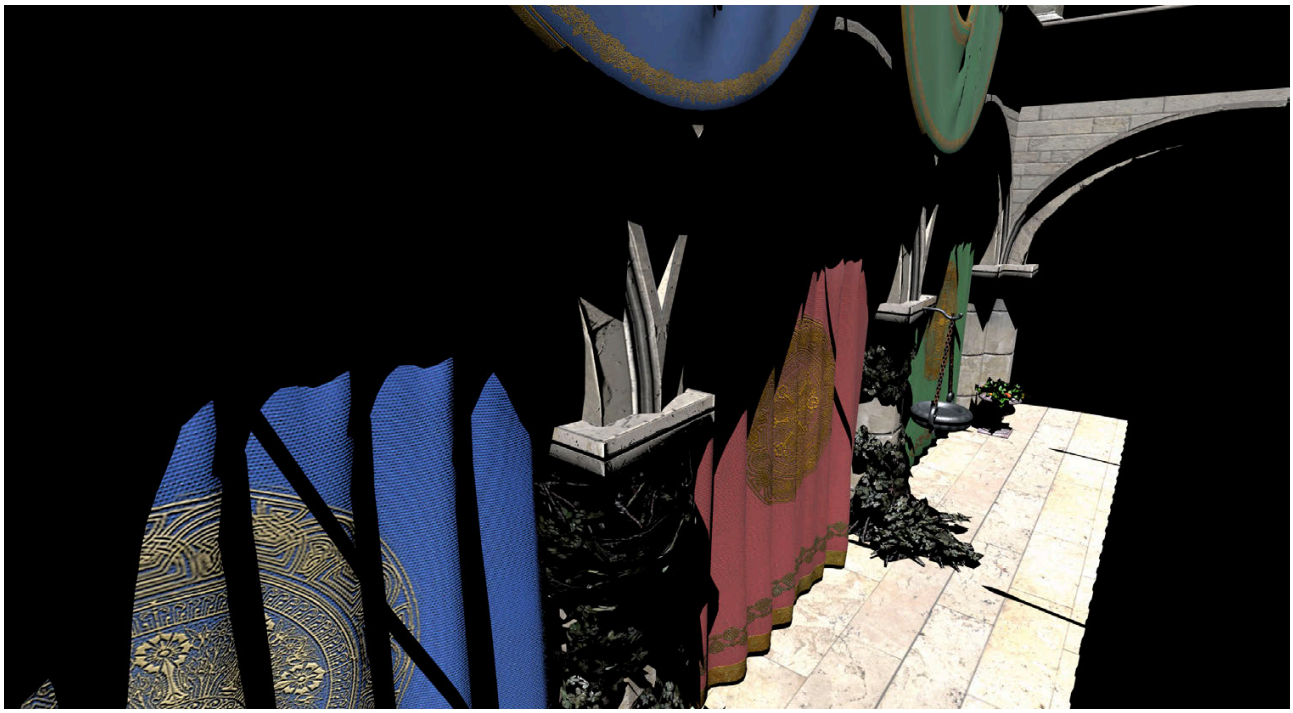
(b) PBRT render. 2048 samples per pixel at 1200×760 resolution. 1878.8 seconds of render time.

The scene has been adjusted to exclude layered materials, which have different models in each renderer. Texture coordinates are flipped in some cases.

Figure A.2: Comparison of a single-bounce path-traced image with an image from the rasteriser. Scene credit McGuire [5].



(a) Rasteriser



(b) Path tracer with a single bounce (no indirect)

The images closely match, with the most apparent differences being in shadow handling.

Appendix B

Implementation Frameworks

The techniques described in this thesis were predominantly implemented and tested in the 3D engine and renderer *LlamaEngine*, using either AMD's RadeonRays [15] or Apple's Metal Performance Shaders [47] libraries for ray intersection. This appendix will provide some background detail on LlamaEngine, its frame-graph rendering system, and the integration of that system with RadeonRays and Metal Performance Shaders.

B.1 LlamaEngine

LlamaEngine (so named because of its initial intended role as the engine for the game *Interdimensional Llama*) is a 3D, physically-based game engine written primarily in the Swift programming language (Apple Inc., Lattner et. al. 2014). Development initially began in late 2016 and has been continued since by myself and Joseph Bennett.

Largely due to my familiarity with the code base, I chose LlamaEngine as the core framework for implementing the GPU-based path tracer and lightmapper. LlamaEngine, and in particular its renderer, does also have a number of other attractions, including:

- Pervasive use of physically-based units for lighting.
- Use of industry-standard material models: either Lambert diffuse or Disney diffuse and GGX specular with Smith height-correlated visibility [95].

Figure B.1: The LlamaEngine editor

- The extensible and easy-to-use SwiftFrameGraph (Bennett and Roughton) [14] rendering system built upon the render graph concept. [99]
- Forward+ and deferred rasterisation paths using clustered shading [79].
- Support for image-based lights using the split-sum approximation (Karis 2013) [96].

Implementing the path tracer and lightmapper necessitated a range of other changes to the engine. The asset pipeline was reworked, adding support for optional per-mesh lightmap UV and chart channels; this reworked asset pipeline was integrated into the main codebase of the engine (where previously it had been isolated in a separate tool), enabling runtime conversion and importing of various file formats. Support for textured materials was added, enabling more detailed scenes and paving the way for sampling the lightmap texture; support for normal maps was added at the same time. Numerous performance improvements and optimisations were also made, enabling large-scale test scenes approaching in scope the scenes used in offline and high-end real-time rendering.

B.1.1 SwiftFrameGraph

The rendering within LlamaEngine is entirely implemented on top of SwiftFrameGraph (Bennett and Roughton [14]), a cross-platform rendering framework that aims to make implementing graphics algo-

rithms simple and performant. The initial implementation was developed in 2017, and it has since been extended, optimised, and reworked. It underlies the implementation of the lightmap path tracer and most other rendering systems described in this thesis.

SwiftFrameGraph is built upon the concept of *render passes*. Conceptually, a render pass is unit of work that produces one or more GPU resources, possibly reading from other GPU resources in the process. Render passes are separated into three categories – draw, compute, and blit passes – which roughly correspond to separate queues on GPU hardware. A draw render pass may dispatch commands involving rasterisation such as rendering triangles or clearing a render target. Compute render passes may dispatch compute shaders, operating on buffers or random-access textures. Blit passes use the GPU's copy engine; they can copy the contents of a buffer to a texture, or clear a buffer to a specific value, for example.

In the FrameGraph API, the user creates render passes (either using closures or as objects that conform to the `DrawRenderPass`, `ComputeRenderPass`, or `BlitRenderPass` protocols. When the FrameGraph executes, it executes all user-provided render passes, enqueuing commands into a buffer. Then, one of the FrameGraph backends – for example, the Metal backend on macOS or the Vulkan backend on Windows or Linux – will parse that stream of commands and translate them to commands in the underlying 3D API. This is usually done once per frame.

The deferred approach of the FrameGraph incurs some small overhead, but also brings major productivity benefits. All resource tracking is implicit rather than explicitly specified by the programmer; compiling a frame ahead of time allows the backends to see all usages of all resources across a frame and therefore insert barriers, fences, and resource transitions as necessary. In addition, the FrameGraph is able to automatically alias resources to reduce the GPU memory footprint.

On APIs that already provide some resource tracking (such as Metal), the native resource tracking can be disabled, amortising the cost of the FrameGraph's tracking. Given knowledge of each render pass' inputs and outputs (by recording which resources each pass reads from and writes to), passes which perform no useful work can be automatically culled; the programmer may insert a blur pass every frame, for example, without needing to check whether any later passes actually need it.

A per-frame view of rendering work also substantially simplifies resource management. Over the course of a frame, many resources – buffers or textures – are needed only to serve as intermediates for other passes in a frame; a texture might serve as a temporary render target that is then processed by a subsequent pass, for example.

These *transient* resources – resources that only last for a frame at most – are made very straightforward to use by SwiftFrameGraph: within a render pass' `execute` method, textures and buffers can be created just like normal objects, inline with the rendering code. No actual allocation is performed when a resource is created; instead, the resource serves as a handle and descriptor for a GPU-backed resource that the FrameGraph backend will provide later. Since the entire lifetime of the resource is known, memory management is taken care of by the FrameGraph and backends as well; creating numerous buffers or textures in a render pass is effectively free, with the memory and minimal CPU cost only being incurred when the resources are actually used within a frame.

SwiftFrameGraph uses the native shading languages of its backends, relying on third-party shader translation tools if the shaders are to be shared. Since the main development and testing platform was macOS, shaders for this project were written natively in the Metal Shading Language, a variant of C++ and close cousin to OpenCL. At the time of writing, the systems have not been implemented for use on other platforms or the Vulkan rendering backend.

subsubsectionFrameGraph Optimisations

Implementing the GPU path tracer and lightmapper exposed shortcomings in the FrameGraph's implementation with regards to performance. The FrameGraph checks resource usages every time the pipeline state changes or a resource binding changes; this generally means inspecting the shader reflection data to infer the usages for the currently bound resources, although this is not necessary for vertex or index buffers. In rasterisation, most resources are bound once for a render pass, with the only changes being in the vertex and index buffers; in general, draw calls should be sorted so as to minimise pipeline state switches wherever possible to avoid GPU overhead. However, path tracing takes place in a series of compute kernels, where each kernel uses a different set of resources; in my implementation there are commonly more than ten separate kernels within a single bounce of the path tracer. This imposes the need for resource tracking to be highly performant.

To give a sense of the scale of this problem, consider a scene like the Sponza Atrium [5]. In that scene, every mesh has six or seven attached resources (e.g. different index and vertex buffers), and there are around 380 meshes in Sponza; at every dispatch call, that's around 2700 items that need to be processed and have their usages tracked. If there are multiple ray bounces in the scene (e.g. usually up to 20 to 25, although it scales with the path count) then there are around 61,000 items to be updated every frame.

To reduce the overhead on this, I first reworked the resource system in a backwards-compatible way to

make all resources integer handles rather than objects, eliminating the reference-counting overhead associated with Swift objects and optimising for cache locality. This enabled moving a number of hash-table lookups (such as for each resource's usages list) to direct array lookups based on the resource's integer handle.[1]

The speed of querying the reflection data for particular shaders and pipeline states was dramatically improved by storing the reflection data in the format the FrameGraph requests (avoiding runtime conversions) and by using vectorised linear search rather than a hash table.[2]

In combination, this work reduced the frame time from 80ms to 17ms: around 4.7 times faster, and almost enabling a 60FPS frame rate without being CPU limited.

B.2 RadeonRays and RadeonProRender

Rather than architect and build an entire GPU path tracer from scratch, I decided to base my implementation on AMD's open-source *RadeonRays* library and *RadeonProRender* framework. RadeonRays provides mechanisms to provide a scene description to the library (consisting of meshes and transforms) and then perform ray intersection and occlusion queries against that scene. More concretely, RadeonRays provides a series of GPU-optimised ray acceleration structures and a set of kernels that can efficiently traverse those structures. RadeonProRender builds a path tracer on top of RadeonRays, implementing kernels for primary ray generation, shading, and accumulation among others, along with the CPU pipeline necessary for submission of those kernels.

Both RadeonRays and RadeonProRender have implementations on top of OpenCL; RadeonRays also supports Vulkan and Embree (Intel 2012) [100]. Although RadeonRays' existing OpenCL implementation was unsuitable for integration into LlamaEngine (since SwiftFrameGraph supports only Metal on macOS, which was the target development platform), the Metal Shading Language and OpenCL happen to be very similar; in fact, in many cases a compatibility header that redefines a few language keywords is all that is needed to port OpenCL kernels to Metal. As such, the RadeonRays kernels used in LlamaEngine are mostly unmodified from the original source.

[1] Each resource's usages are stored in a linked list, where the linked list nodes are allocated from a per-thread arena which is solely responsible for linked list nodes. This strategy means that resource usage iteration is reasonably cache-friendly and performant while also enabling multi-threaded usage list construction with non-blocking atomic operations.

[2] Linear search was chosen over binary search since the low number of elements to search means that linear search's simplicity outperforms the algorithmic-complexity advantage of binary search.

The implementation of the LlamaEngine path tracer differs more significantly from RadeonProRender. Many of these differences are described in depth in Chapter 3, 5, and 6; however, in general, the porting process for RadeonProRender was more a matter of copying pieces of functionality rather than directly translating the source code, with the hope that applying this process would lead to better understanding of how the implementation works. As such, the CPU implementation of the path tracer within LlamaEngine bears fairly little resemblance to RadeonProRender, although it does perform roughly the same steps in roughly the same order.

RadeonRays also required modifications to the CPU source to support integration with LlamaEngine, brought about by the use of a different underlying API (Metal through SwiftFrameGraph) and the fact that LlamaEngine is predominantly written in Swift, which cannot interoperate with C++ except through C. In cases where Swift code must call into a C++ library, the solution to this is relatively simple: since Swift can interoperate with C, writing a C wrapper around the C++ API is sufficient. However, it is slightly more difficult when C++ code needs to call into Swift code.

RadeonRays is structured with a different backend for each platform (such as OpenCL) it supports. Generally, a RadeonRays backend translates the RadeonRays calls (which may commonly be commands to bind GPU resources or to dispatch a certain kernel) into commands for the underlying API. The pre-existing backends within RadeonRays are all written in C++ and follow an inheritance structure; that is, each backend overrides a number of virtual methods in a set of concrete subclasses. For use in LlamaEngine, the goal was to implement a new backend on top of SwiftFrameGraph, which would in turn translate to Metal calls. Rather than implementing a C API on top of SwiftFrameGraph and building a RadeonRays backend on top of that, I decided the best path was to implement a generic function-pointer based backend for RadeonRays, enabling other arbitrary backends to simply provide an implementation for each of the required functions. These function pointers are filled by Swift code within LlamaEngine.

B.2.1 RadeonRays and Render Graph Resources

SwiftFrameGraph introduces the concept of transient resources, which have a one-frame lifetime and are automatically managed by the SwiftFrameGraph backend. These have no matching equivalent within RadeonRays; instead, RadeonRays either uses persistent buffers or dequeues buffers from pools for reuse between frames. When integrating RadeonRays with SwiftFrameGraph, one goal was for RadeonRays to smoothly interoperate with SwiftFrameGraph's resource system; in effect, this means that transient

buffers must be able to be created within LlamaEngine and passed to RadeonRays, which may in turn bind those buffers during the course of a frame.

To enable this, the RadeonRays function-pointer-based backend has the ability to import references to external resources.[3] When those resources are exported to RadeonRays from within LlamaEngine, a callback is also introduced to delete RadeonRays' references to those resources at the end of the frame.

However, not every resource is transient; some, such as the ray acceleration data structures, are owned by RadeonRays and are persistent. These resources can be created by a call to LlamaEngine's implementation of the `createBuffer` function by RadeonRays. When RadeonRays requests that a resource be deleted, the LlamaEngine implementation checks that RadeonRays owns (was responsible for the creation of) that resource before disposing it; the dispose callback is called by RadeonRays even for externally-created resources, but in the case of transient FrameGraph resources has no effect.

One final complication is that RadeonRays may attempt modify its persistent resources while they are in use by the GPU. This occurs when, for example, an object is moved in the scene, requiring a rebuild of the ray acceleration structure. To avoid a resource hazard, the implementation of the FrameGraph was modified to track when resources are in use by the GPU and to block work submission until the GPU has finished accessing those resources. This is done on a per-frame-submission granularity; each resource tracks which frame it was most recently read from and written to by the GPU, and then CPU access waits until it is safe to access the resource (i.e. all GPU writes have finished if the CPU is accessing the resource as a read, or all GPU reads have finished if the CPU is accessing as a write). This does introduce a pipeline stall, since the GPU is effectively idle in the time between the CPU accessing the resource and the CPU submitting a new frame after that access. However, since these modifications are done infrequently and always in response to user action, the incurred overhead is minimal and preferable to the extra GPU memory usage that would be entailed by multiple buffering.

If animation or rapidly-changing scenes were required, it may be worthwhile to use multiple buffering for the ray acceleration structure, or alternatively to split the scene into a single-buffered static structure and a multi-buffered dynamic structure that may be updated independently. For lightmap baking, the scene must be static; therefore, only a single acceleration structure that is immutable during the baking process is needed.

[3] The ability to import externally-created buffers already existed within the Vulkan and OpenCL backends.

B.3 Metal Performance Shaders

In June of 2018, Apple added support for GPU-based ray intersection to its Metal Performance Shaders (MPS) framework. At this stage, RadeonRays was already integrated into LlamaEngine and the core path tracing framework was in place; however, I considered it worthwhile to integrate the Metal Performance Shaders implementation for sake of performance comparison. It seemed likely that the implementation would be highly optimised for both the Metal API's implementation and the specific test hardware.[4]

In practice, the Metal Performance Shaders implementation is not a clear win over the RadeonRays kernels in terms of performance; in some cases performance is noticeably poorer, while in others it holds a significant advantage; Table B.1 demonstrates this on a range of scenes. It does, however, raise a number of implementation considerations.

The most broadly relevant of these is the requirement that all vertices be contained within a single buffer, and in particular that those vertices be pre-transformed and placed into a flat hierarchy for best performance, as if they all belong to a single mesh. The API also provides for instanced rendering in a two-level hierarchy for frequently-updated scenes or scenes where the memory overhead of duplicating vertices is unacceptable; however, this instanced rendering path carries significant overhead, and in the case of a static scene (as lightmap baking necessitates) is best avoided.[5] The effect of this is that we lose the concept of a shape or instance index; instead, all that is provided for us with each intersection is the primitive index, barycentric coordinates, and intersection distance.[6] We therefore need to be able to convert that whole-scene-mesh primitive index into the index of the mesh within the scene.[7]

It would be theoretically possible to place all other attributes for all vertices into a separate, linear buffer, on top of the positions and primitive indices MPS already requires. However, there is a large amount of per-vertex data that may be accessed within the shading kernel – surface normals, tangent and bitangents, and texture coordinates, to name a few – and the cost of duplicating that data for every vertex is significant.

When we generate the position buffer for the full scene – that is, the buffer containing the transformed positions of every vertex in the scene –, we also generate its index buffer, copying from each mesh's index

[4] At the WWDC 2018 conference, Apple demonstrated the Metal Performance Shaders ray intersection kernels on AMD Vega 56 hardware, which is architecturally identical to the Vega Pro 64 used predominantly for testing.

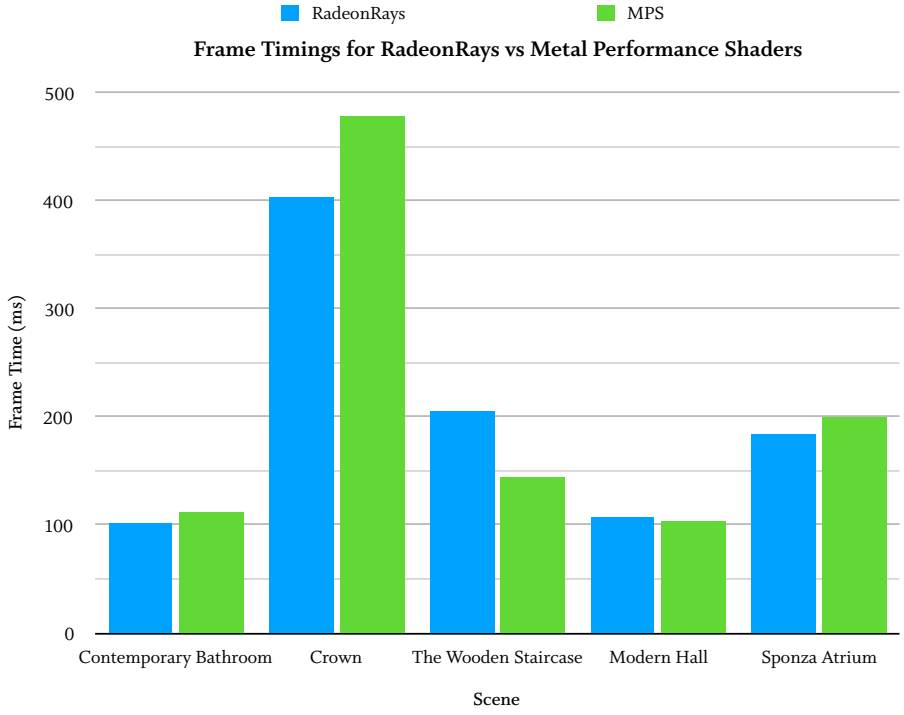
[5] RadeonRays provides similar options of a flat acceleration structure or a two level hierarchy, although in RadeonRays' case the differences between the two are hidden for consumers of the API.

[6] MPS intersections *can* also carry the instance index; however, this is only relevant when a two-level hierarchy is being used.

[7] This index is used to index into an argument buffer array containing all meshes in the scene, as is described in Section B.4.

Table B.1: Timing per frame for RadeonRays vs. Metal Performance Shaders.

Scene	RadeonRays	MPS
Contemporary Bathroom	$101ms$	$111ms$
Crown	$403ms$	$478ms$
The Wooden Staircase	$205ms$	$144ms$
Modern Hall	$106ms$	$104ms$
Sponza Atrium	$183ms$	$199ms$



Lower is better. Results were averaged over 1000 frames.
Ray direction sorting is disabled on the RadeonRays implementation.
Scenes can be viewed in Appendix D.

buffer. As we add each mesh instance, we can record its starting index within the full-scene index buffer; for example, one mesh might start at index 300, which would map to primitive 100 (since all primitives supported are triangles). From this, we can generate a buffer containing the starting primitive offset for every mesh in the scene.

Upon receiving an intersection result from MPS, we can retrieve the primitive index and use it to binary search within the starting primitive offset buffer, returning the index i of the last element in that buffer which is at most the primitive index. i is therefore the mesh instance index, matching the shape or mesh instance index that RadeonRays would return us; the primitive index within that mesh instance is given by subtracting the primitive offset buffer's element for i from the primitive index returned by MPS.

Alternatively, we can avoid the binary search at the cost of extra memory usage by instead including a per-primitive stream of mesh indices. The mesh index i is found by looking up the value in the per-primitive buffer at the whole-scene primitive index; the whole-scene primitive index is then mapped to the mesh instance primitive index in the same way as per the binary search method. Using this extra index buffer yields a roughly 2% overall performance improvements in my tests; whether it is worthwhile will therefore depend on whether the increased memory usage offsets the lower ALU cost and potentially lower bandwidth requirements.[8]

To abstract over the API differences between MPS and RadeonRays, I implemented a number of functions within the shader code that map from each API's intersection data types to a common intermediate. These functions cast from untyped pointers to the API type, where the API is fixed at compile-time of the shader by way of function constants.[9] In addition, I modified RadeonRays' `Ray` structure to match that required by MPS; this simply involved changing the layout of some fields.

B.4 Resource Binding

In path tracing it is not known ahead of time what meshes will be intersected with when a batch of rays is fired; this is in contrast to rasterisation, where the only resources needed at the time of a draw call are those associated with the current draw. This necessitates having the entire scene description – every

[8] When using an index buffer, each thread needs to fetch two elements: the mesh index and the base primitive offset for that mesh index. With the binary search, the worst case is $\log_2(\mathit{meshCount}) + 1$ fetches, but all threads will be fetching from the same relatively small region in memory.

[9] Function constants (also known as specialisation constants) are lightweight compile-time constants that can be used to change the behaviour within a shader without incurring runtime costs.

mesh, with all of its associated textures, skinning matrices, and vertex streams – available when shading the scene after every ray batch.

This is implemented in LlamaEngine (Appendix B.1) using Metal's *argument buffers* (analogous to Vulkan descriptor sets). Conceptually, an argument buffer is a struct containing some inline data (the material type or the skinning matrix, for example) and references to other resources such as buffers or textures. These structs can be placed in an array, where the array is indexed by the shape or instance index. The scene can therefore be described as an array of `Mesh` structs. The Metal shader description of `Mesh` for LlamaEngine has the following structure:

Listing B.1: Metal Shading Language Mesh Description

```
struct Mesh {
  float3x4 meshToWorldTranspose [[ id(0) ]];
  const device float3 *positions;
  const device float4 *normalsAndTexCoords; // packed_float3 and then a half2
  const device uint *tangentsAndBitangents; // 2 short3Normalized
  const device uint *lightmapUVs; // ushort2Normalized
  const device uint *lightmapCharts;
  const device ushort *indices; // may also be uint* if the mesh has 32-bit indices
  const device ushort *lightmapIndices; // the lightmap-specific index buffer.

  const device float4 *lightmapChartScalesAndOffsets;

  LightingBRDF lightingBRDF;
  IndexType indexType;
  IndexType lightmapIndexType;

  uint16_t materialIndex;
  uint8_t materialTranslucency;

  uint8_t hasVertexColours;
  uint8_t hasTangentsAndBitangents;
};
```

The material index indexes into a separate buffer of materials, which in turn contains references to textures used in those materials.

In Metal Shading Language, pointers within an argument buffer struct are references to other buffers. Note that use of argument buffers in Metal requires manual residency and hazard tracking for all resources referenced within the argument buffer; this is performed automatically by the SwiftFrameGraph framework.

Appendix C

Image Gallery: Reconstruction Error from Ambient Dice vs. Spherical Gaussians

This chapter contains a comparison of the Ambient Dice SRBF and spherical Gaussian encoding formats on a range of environment maps from Debevec [88] and Vogl [85]. All images and error metrics were generated using a modified version of the open source Probulator tool [87].

The images are clipped to the upper hemisphere to represent lightmap use cases. For radiance and Lambertian irradiance, this means the sample direction (i.e. the surface normal in the case of the irradiance) is restricted to the upper hemisphere; for GGX specular, the normal is fixed at $(0, 0, 1)$ and instead the image is parameterised by the viewing direction.

The encoding methods presented are:

- Monte Carlo Importance Sampling (MCIS): the importance-sampled ground truth. Heitz's method [72] is used for GGX, while a cosine-weighted hemisphere is sampled for Lambertian diffuse.
- AD9: Nine Ambient Dice cosine-variant lobes are arranged as described in Section 8.3.1.
- AD12: Twelve Ambient Dice cosine-variant lobes are arranged at the vertices of an icosahedron.
- SG9: Nine spherical Gaussian lobes with λ =6 are arranged on the upper hemisphere according to Vogel's sphere [87].
- SG12: Twelve spherical Gaussian lobes with $\lambda = 6$ are arranged on the upper hemisphere according to Vogel's sphere [87].

All basis functions are solved to minimise least-squares error over the sphere (rather than the hemisphere). λ of 6 was used due to its being Probulator's chosen default for spherical Gaussians.

The custom fits for diffuse and specular presented in Section 8.4 are used for cosine-lobe Ambient Dice. Spherical Gaussians use the anisotropic fit for specular and Hill's fit for diffuse [90, 101].

HDR images are tone-mapped and gamma corrected; therefore, intensities in the images do not increase linearly but more closely match the perceived intensity. Negative values are clamped to zero.

The RMSE is the root-mean-squared error, averaged over all colour channels, between the true intensity value (from MCIS) and the approximation. Since the least-squares encoding process minimises the mean-squared error, the root-mean-squared error is the correct metric to gauge the accuracy of the fit, and reasonable for determining the BRDF reconstruction quality.

The RMSE is not normalised in any way; if the RMSE decreases with higher roughness values the overall intensity is likely decreasing. As such, the RMSE should only be compared between different basis functions for the same BRDF (along rows).

In general, both variants of Ambient Dice present more accurate reconstruction than spherical Gaussians for rough and diffuse materials, although spherical Gaussians are often more accurate for low roughness values. Interestingly, AD9 sometimes outperforms AD12; when this occurs, it is likely because the AD9 lobes are rotated to better align with the scene light sources.

Figure C.1: Ambient Dice vs. Spherical Gaussian Reconstruction on 'Pisa'

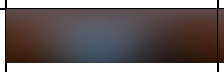
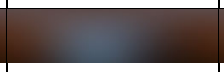
	MCIS	AD9	AD12	SG9	SG12
Radiance					
RMSE		0.297	0.285	0.264	0.235
Lambert					
RMSE		0.00755	0.00580	0.0150	0.0138
GGX, $\alpha = 0.1$					
RMSE		0.145	0.133	0.109	0.076
GGX, $\alpha = 0.2$					
RMSE		0.0749	0.0698	0.0542	0.0404
GGX, $\alpha = 0.4$					
RMSE		0.0158	0.0202	0.0353	0.0378
GGX, $\alpha = 0.6$					
RMSE		0.00869	0.0119	0.0388	0.0391

Figure C.2: Ambient Dice vs. Spherical Gaussian Reconstruction on 'Ennis'

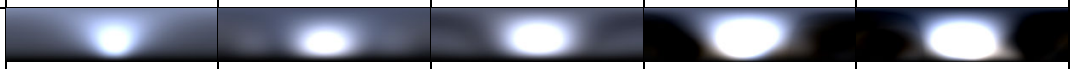
	MCIS	AD9	AD12	SG9	SG12
Radiance					
RMSE		5.25	5.83	6.45	5.54
Lambert					
RMSE		0.148	0.105	0.327	0.225
GGX, $\alpha = 0.1$					
RMSE		2.17	2.74	2.97	2.34
GGX, $\alpha = 0.2$					
RMSE		0.947	1.37	1.47	1.12
GGX, $\alpha = 0.4$					
RMSE		0.276	0.409	0.566	0.574
GGX, $\alpha = 0.6$					
RMSE		0.130	0.137	0.378	0.466
GGX, $\alpha = 0.8$					
RMSE		0.119	0.091	0.299	0.374

Figure C.3: Ambient Dice vs. Spherical Gaussian Reconstruction on 'Grace'

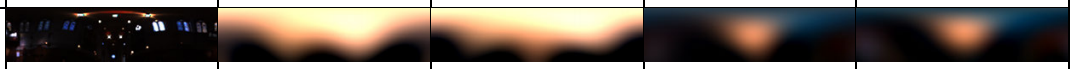
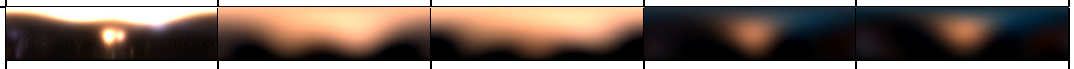
	MCIS	AD9	AD12	SG9	SG12
Radiance					
RMSE		27.6	27.6	27.7	27.7
Lambert					
RMSE		0.0118	0.0130	0.300	0.303
GGX, $\alpha = 0.1$					
RMSE		0.956	0.936	1.22	1.22
GGX, $\alpha = 0.2$					
RMSE		0.395	0.377	0.670	0.671
GGX, $\alpha = 0.4$					
RMSE		0.0809	0.0733	0.316	0.319
GGX, $\alpha = 0.6$					
RMSE		0.0178	0.0170	0.191	0.192
GGX, $\alpha = 0.8$					
RMSE		0.0127	0.0144	0.121	0.122

Figure C.4: Ambient Dice vs. Spherical Gaussian Reconstruction on 'Uffizi'

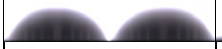
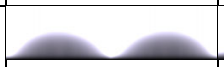
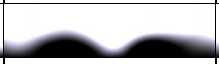
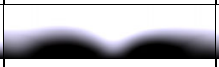
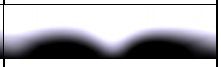
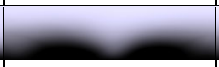
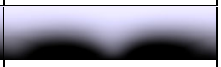
	MCIS	AD9	AD12	SG9	SG12
Radiance					
RMSE		3.96	4.06	3.67	3.25
Lambert					
RMSE		0.0367	0.0397	0.0662	0.0573
GGX, $\alpha = 0.1$					
RMSE		2.01	2.10	1.77	1.41
GGX, $\alpha = 0.2$					
RMSE		1.04	1.10	0.827	0.680
GGX, $\alpha = 0.4$					
RMSE		0.270	0.291	0.411	0.455
GGX, $\alpha = 0.6$					
RMSE		0.0691	0.0735	0.299	0.323
GGX, $\alpha = 0.8$					
RMSE		0.0323	0.0300	0.198	0.207

Figure C.5: Ambient Dice vs. Spherical Gaussian Reconstruction on 'Wells'

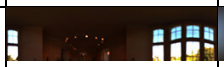
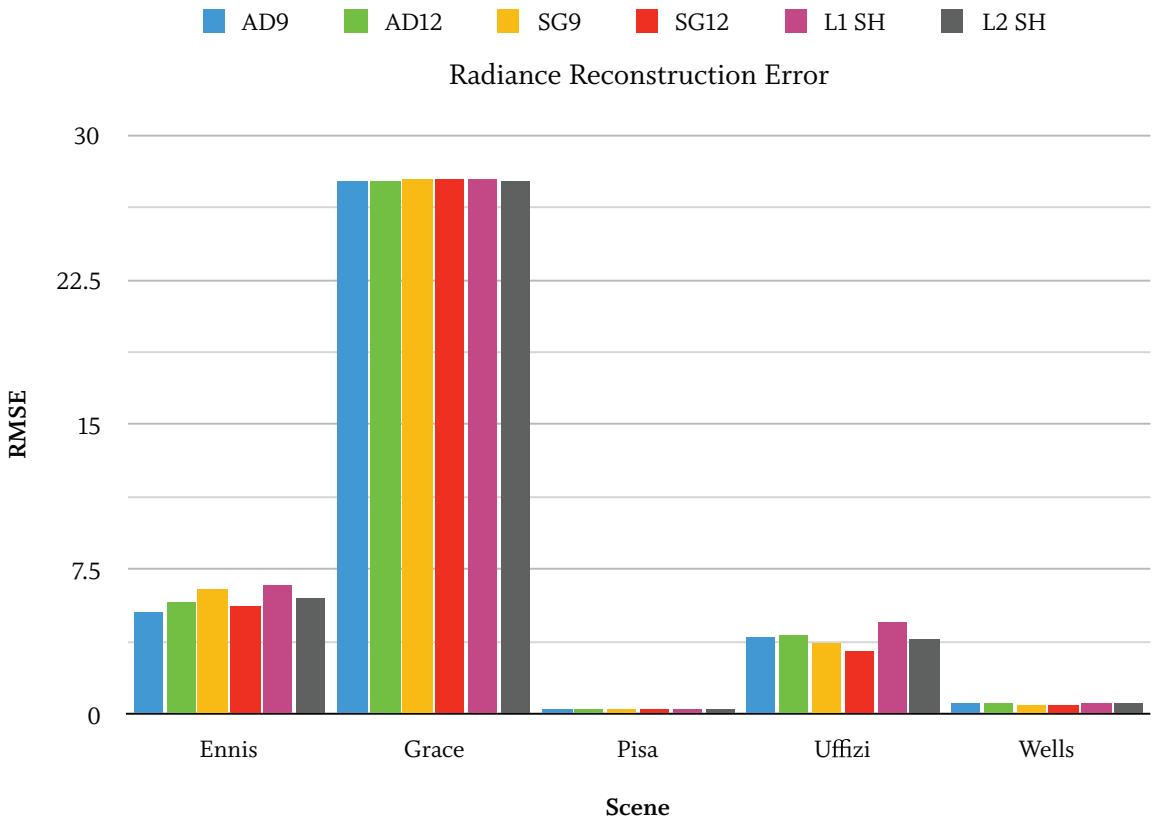
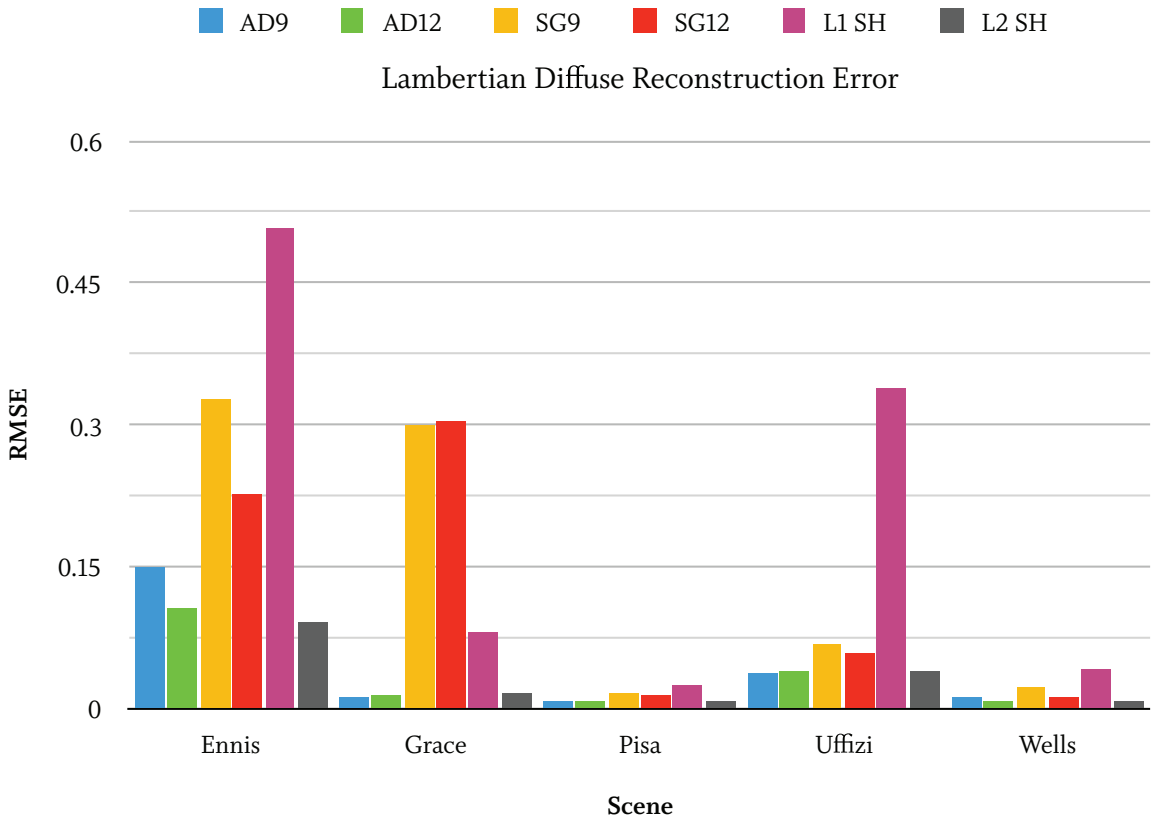
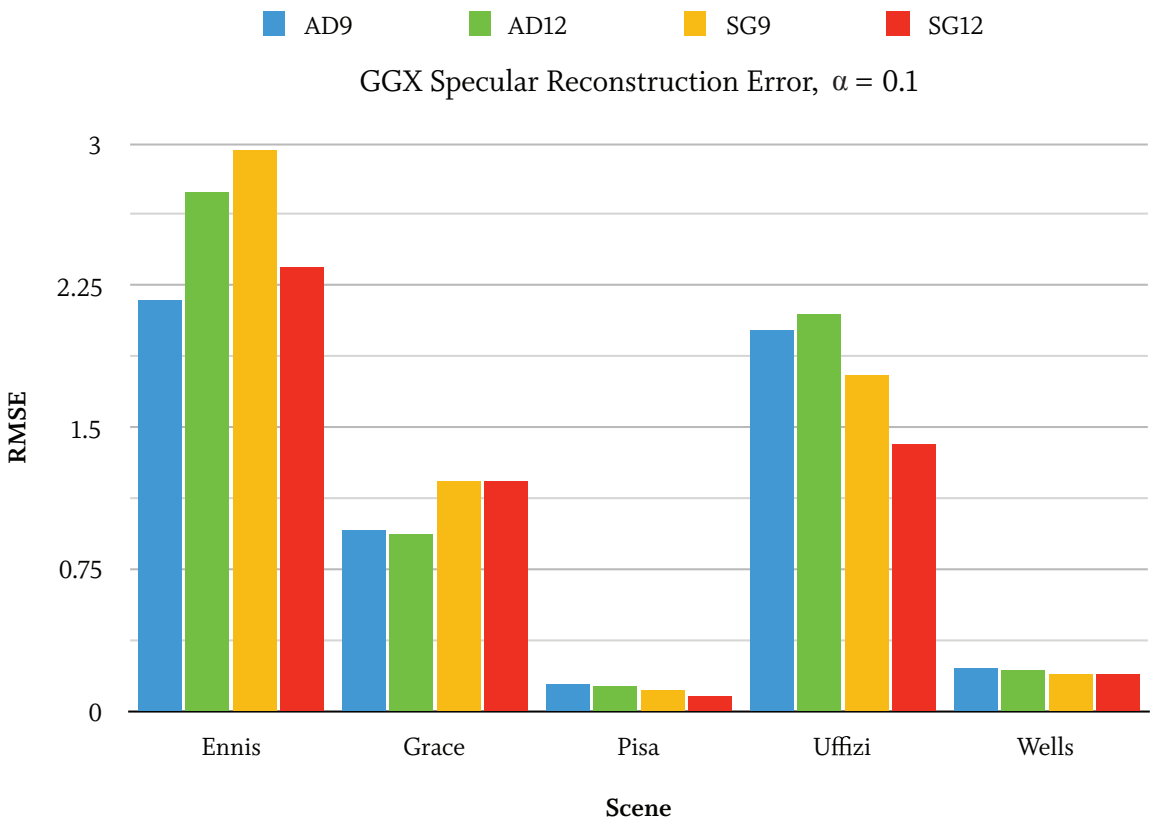
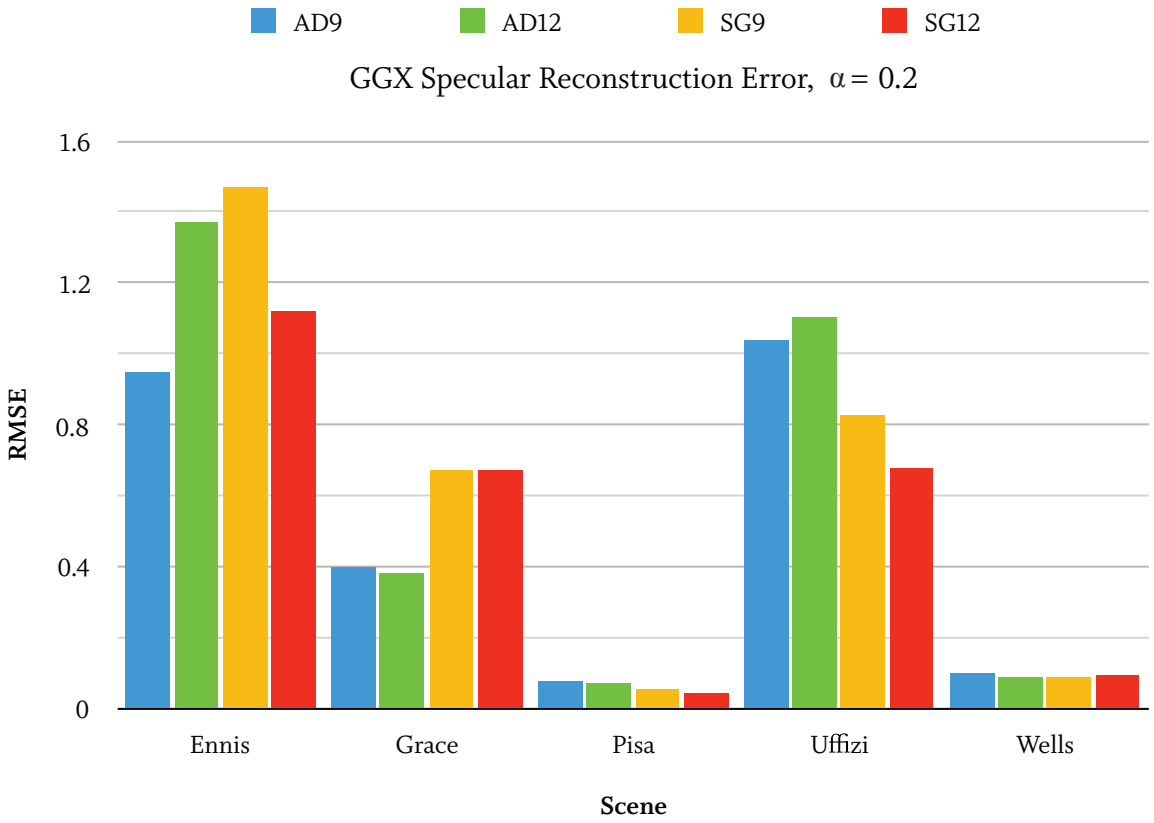
	MCIS	AD9	AD12	SG9	SG12
Radiance					
RMSE		0.5574	0.5492	0.5283	0.5210
Lambert					
RMSE		0.0110	0.00729	0.0217	0.011408
GGX, $\alpha = 0.1$					
RMSE		0.227	0.219	0.195	0.192
GGX, $\alpha = 0.2$					
RMSE		0.0955	0.0894	0.0848	0.0948
GGX, $\alpha = 0.4$					
RMSE		0.0169	0.0176	0.0574	0.0617
GGX, $\alpha = 0.6$					
RMSE		0.00964	0.0118	0.0537	0.0526

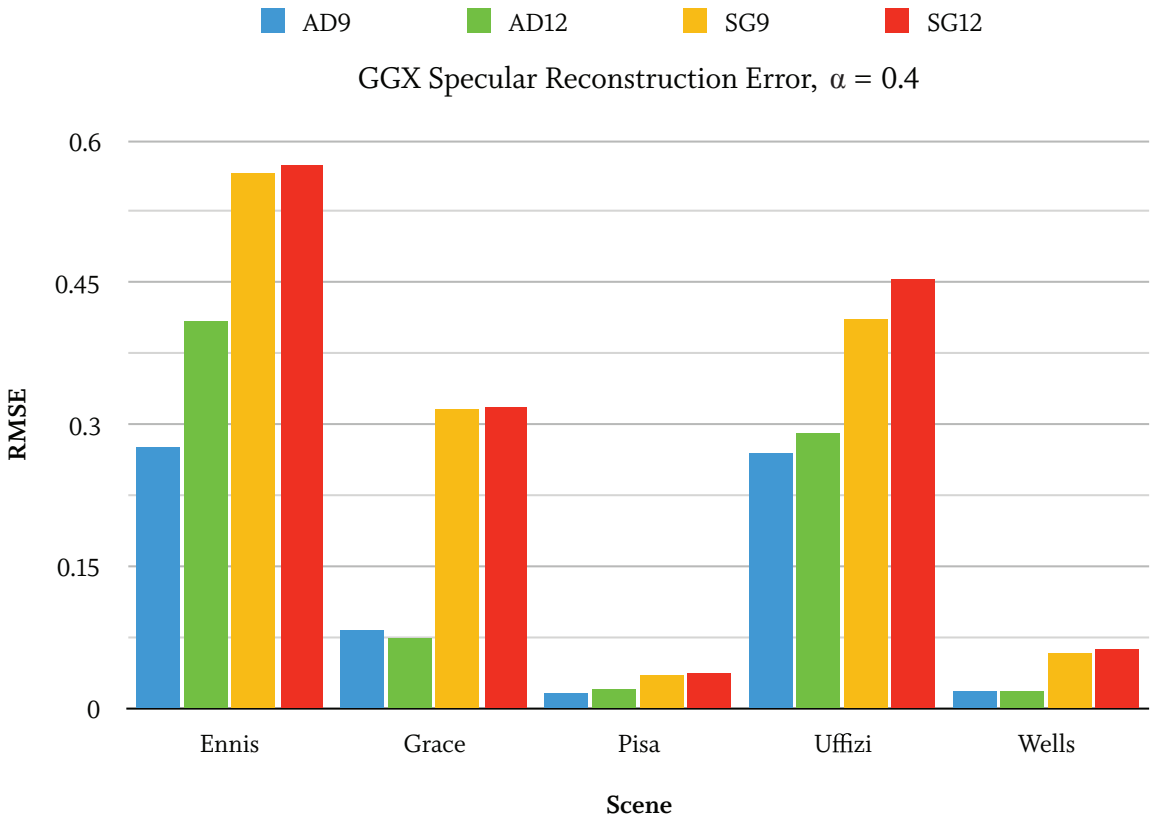
Figure C.6: Comparison graphs of reconstruction error for spherical harmonics, spherical Gaussians, and cosine-lobe Ambient Dice

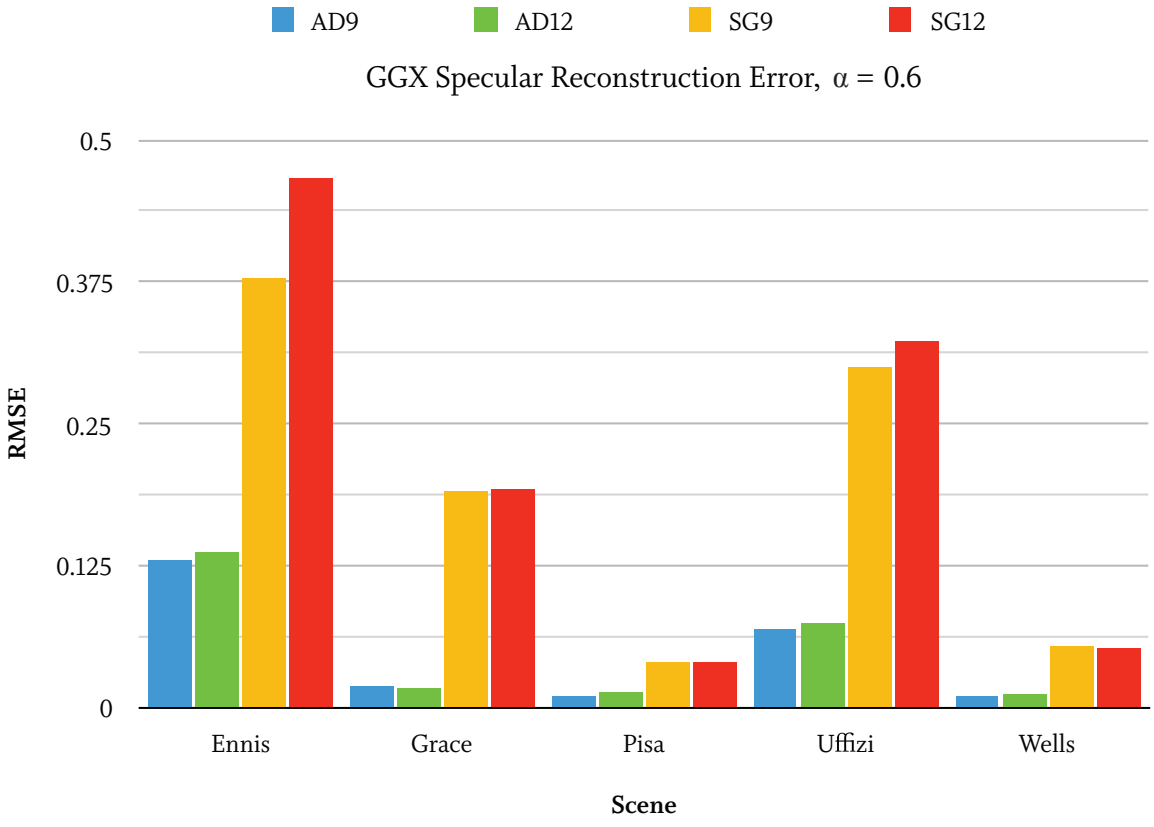





AD9
AD12
SG9
SG12
GGX Specular Reconstruction Error, α = 0.1
RMSE
3
2.25
1.5
0.75
0
Ennis
Grace
Pisa
Uffizi
Wells
Scene


AD9
AD12
SG9
SG12
GGX Specular Reconstruction Error, α = 0.2
RMSE
1.6
1.2
0.8
0.4
0
Ennis
Grace
Pisa
Uffizi
Wells
Scene


AD9
AD12
SG9
SG12
GGX Specular Reconstruction Error, α = 0.4
RMSE
0.6
0.45
0.3
0.15
0
Ennis
Grace
Pisa
Uffizi
Wells
Scene


AD9
AD12
SG9
SG12
GGX Specular Reconstruction Error, α = 0.6
RMSE
0.5
0.375
0.25
0.125
0
Ennis
Grace
Pisa
Uffizi
Wells
Scene

Appendix D

Image Gallery: Test Scenes

This appendix contains reference images for scenes used within this thesis. All images were rendered within the LlamaEngine (Appendix B.1) path tracer built for this thesis.

Figure D.1: 'Contemporary Bathroom' (Mareck) [40]

Figure D.2: 'Crown' (Lubich) [40]

Figure D.3: 'The Wooden Staircase' (Wig42) [5, 59]

Figure D.4: 'Modern Hall' (NewSee2l035) [5, 59]

Figure D.5: 'Sponza Atrium' (Meinl, McGuire) [5]

Bibliography

[1] E. Haines and T. Akenine-Möller, Eds., *Ray Tracing Gems*. Apress, 2019. [Online]. Available: http://raytracinggems.com

[2] 'NVIDIA Announces RTX Technology: Real Time Ray Tracing Acceleration for Volta GPUs and Later,' 2018. [Online]. Available: https://www.anandtech.com/show/12546/nvidia-unveils-rtx-technology-real-time-ray-tracing-acceleration-for-volta-gpus-and-later

[3] D. Flatt, 'AMD Radeon Rays Integrated into Unity's GPU Progressive Lightmapper,' 2018. [Online]. Available: https://blogs.unity3d.com/2018/03/29/amd-radeon-rays-integrated-into-unitys-gpu-progressive-lightmapper/ (Accessed 2018-06-14).

[4] S. Hillaire, *Real-Time Raytracing for Interactive Global Illumination Workflows in Frostbite*. Electronic Arts, 2018, presented at GDC 2018. [Online]. Available: https://www.ea.com/frostbite/news/real-time-raytracing-for-interactive-global-illumination-workflows-in-frostbite

[5] M. McGuire. (2017, July) Computer Graphics Archive. [Online]. Available: https://casual-effects.com/data (Accessed 2019-01-14).

[6] P.-P. Sloan and A. Silvennoinen, 'Directional lightmap encoding insights,' in *SIGGRAPH Asia 2018 Technical Briefs*. ACM, 12 2018, pp. 1--3. doi:10.1145/3283254.3283281

[7] H. Chen and X. Liu, 'Lighting and Material of Halo 3,' in *ACM SIGGRAPH 2008 Games*, ser. SIGGRAPH '08. New York, NY, USA: ACM, 2008, pp. 1--22. doi:10.1145/1404435.1404437

[8] *Precomputed Global Illumination in Frostbite*, presented at GDC 2018. [Online]. Available: https://www.gdcvault.com/play/1025434/Precomputed-Global-Illumination-in

[9] J. Wang, P. Ren, M. Gong, J. Snyder, and B. Guo, 'All-frequency Rendering of Dynamic, Spatially--varying Reflectance,' *ACM Trans. Graph.*, vol. 28, no. 5, p. 133:1–133:10, Dec. 2009.

[10] D. Neubelt and M. Pettineo, 'Physically Based Shading in Theory and Practice: Advanced Lighting R&D at Ready At Dawn Studios,' 2017, presented at SIGGRAPH 2015. [Online]. Available: https://blog.selfshadow.com/publications/s2015-shading-course/

[11] S. Laine, T. Karras, and T. Aila, 'Megakernels considered harmful,' *Proceedings of the 5th High-Performance Graphics Conference on - HPG 13*, 2013. [Online]. Available: https://research.nvidia.com/publication/megakernels-considered-harmful-wavefront-path-tracing-gpus. doi:10.1145/2492045.2492060

[12] M. Iwanicki and P.-P. Sloan, 'Ambient Dice,' in *Eurographics Symposium on Rendering - Experimental Ideas & Implementations*, 2017.

[13] M. Pharr, W. Jakob, and G. Humphreys, *Physically Based Rendering: from Theory to Implementation*, 3rd ed. Morgan Kaufmann, 2017.

[14] Roughton, Thomas and Bennett, Joseph, 'SwiftFrameGraph,' 2017. [Online]. Available: https://github.com/troughton/SwiftFrameGraph

[15] 'RadeonRays SDK.' [Online]. Available: https://github.com/GPUOpen-LibrariesAndSDKs/RadeonRays_SDK (Accessed 2018-05-14).

[16] A. Gareffa, 'PS5 powered by Navi in 2020, AMD making Navi with Sony input,' Jun 2018. [Online]. Available: https://www.tweaktown.com/articles/8643/ps5-powered-navi-2020-amd-making-sony-input/index.html

[17] Cyan, 'Myst,' 1993.

[18] Pettineo, Matt, 'The Baking Lab.' [Online]. Available: https://github.com/TheRealMJP/BakingLab (Accessed 2018-10-26).

[19] F. Sanglard, 'Quake 2 Source Code Review,' 2011. [Online]. Available: http://fabiensanglard.net/quake2/quake2_software_renderer.php (Accessed 2019-01-13).

[20] id Software, 'Quake,' 1996.

[21] J. Bush, 'Quake Lightmaps,' 2015. [Online]. Available: https://jbush001.github.io/2015/06/11/quake-lightmaps.html (Accessed 2018-11-08).

[22] M. Iwanicki, 'Lighting technology of The Last of Us,' *ACM SIGGRAPH 2013 Talks on - SIGGRAPH 13*, 2013. doi:10.1145/2504459.2504484

[23] I. Castaño. (2010) Hemicube Rendering and Integration. [Online]. Available: http://the-witness.net/news/2010/09/hemicube-rendering-and-integration/ (Accessed 2018-04-12).

[24] G. Greger, P. Shirley, P. M. Hubbard, and D. P. Greenberg, 'The Irradiance Volume,' *IEEE Comput. Graph. Appl.*, vol. 18, no. 2, pp. 32--43, Mar. 1998.

[25] N. Tatarchuk, *Irradiance Volumes for Games*, 2005, presented at GDC 2005. [Online]. Available: https://developer.amd.com/wordpress/media/2012/10/Tatarchuk_Irradiance_Volumes.pdf

[26] J. Cohen, M. Olano, and D. Manocha, 'Appearance-preserving Simplification,' in *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '98. New York, NY, USA: ACM, 1998, p. 115–122. doi:10.1145/280814.280832

[27] R. Ramamoorthi and P. Hanrahan, 'An efficient representation for irradiance environment maps,' *Proceedings of the 28th annual conference on Computer graphics and interactive techniques - SIGGRAPH 01*, 2001. doi:10.1145/383259.383317

[28] id Software, 'Quake III Arena,' 1999. [Online]. Available: https://github.com/id-Software/Quake-III-Arena

[29] D. Lazarov, 'Physically-based lighting in Call of Duty: Black Ops,' in *SIGGRAPH 2011 Course: Advances in Real-Time Rendering in Games*. [Online]. Available: http://advances.realtimerendering.com/s2011/Lazarov-Physically-Based-Lighting-in-Black-Ops%20(Siggraph%202011%20Advances%20in%20Real-Time%20Rendering%20Course).pptx

[30] J. Wang, M. Gong, J. Snyder, B. Guo, and P. Ren, 'All-Frequency Rendering of Dynamic, Spatially-Varying Reflectance,' *ACM Transactions on Graphics*, January 2007. [Online]. Available: https://www.microsoft.com/en-us/research/publication/all-frequency-rendering-of-dynamic-spatially-varying-reflectance/

[31] Ready at Dawn, 'The Order: 1886,' 2015.

[32] 'Metal Feature Set Tables.' [Online]. Available: https://developer.apple.com/metal/Metal-Feature-Set-Tables.pdf (Accessed 2019-01-13).

[33] C. M. Goral, K. E. Torrance, D. P. Greenberg, and B. Battaile, 'Modeling the interaction of light between diffuse surfaces,' *Proceedings of the 11th annual conference on Computer graphics and interactive techniques - SIGGRAPH 84*, 1984. doi:10.1145/800031.808601

[34] M. F. Cohen and D. P. Greenberg, 'Tutorial: Computer graphics; image synthesis,' K. I. Joy, C. W. Grant, N. L. Max, and L. Hatfield, Eds. New York, NY, USA: Computer Science Press, Inc., 1988, ch. The Hemi-cube; a Radiosity Solution for Complex Environments, pp. 254--263. [Online]. Available: http://dl.acm.org/citation.cfm?id=95075.95129

[35] P. Christensen, J. Fong, J. Shade, W. Wooten, B. Schubert, A. Kensler, S. Friedman, C. Kilpatrick, C. Ramshaw, M. Bannister, B. Rayner, J. Brouillat, and M. Liani, 'RenderMan: An Advanced Path--Tracing Architecture for Movie Rendering,' *ACM Trans. Graph.*, vol. 37, no. 3, p. 30:1–30:21, Aug. 2018.

[36] B. Burley, D. Adler, M. J.-Y. Chiang, H. Driskill, R. Habel, P. Kelly, P. Kutz, Y. K. Li, and D. Teece, 'The Design and Evolution of Disney's Hyperion Renderer,' *ACM Transactions on Graphics*, vol. 37, no. 3, Aug. 2018. doi:10.1145/3182159

[37] L. Fascione, J. Hanika, M. Leone, M. Droske, J. Schwarzhaupt, T. Davidovic, A. Weidlich, and J. Meng, 'Manuka: A Batch-Shading Architecture for Spectral Path Tracing in Movie Production,' *ACM Trans. Graph.*, vol. 37, no. 3, p. 31:1–31:18, Aug. 2018.

[38] M. Lee, B. Green, F. Xie, and E. Tabellion, 'MoonRay: Vectorized Production Path Tracing,' in *Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics*, V. Havran and K. Vaiyanathan, Eds. ACM, 2017. doi:10.1145/3105762.3105768

[39] J. M. Hammersley and D. C. Handscomb, *Monte Carlo Methods*. Methuen, 1964.

[40] W. J. Matt Pharr and G. Humphreys, 'Scenes for pbrt-v3.' [Online]. Available: https://pbrt.org/scenes-v3.html (Accessed 2019-01-02).

[41] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan, 'Ray Tracing on Programmable Graphics Hardware,' in *ACM SIGGRAPH 2005 Courses*, ser. SIGGRAPH '05. New York, NY, USA: ACM, 2005. doi:10.1145/1198555.1198798

[42] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich, 'OptiX: A General Purpose Ray Tracing Engine,' *ACM Transactions on Graphics*, August 2010.

[43] 'Radeon ProRender.' [Online]. Available: https://github.com/GPUOpen-LibrariesAndSDKs/RadeonProRender-Baikal (Accessed 2018-05-14).

[44] 'Introduction to the NVIDIA Turing Architecture.' [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf (Accessed 2018-04-25).

[45] M. Sandy, 'Announcing Microsoft DirectX Raytracing!' 2018. [Online]. Available: https://blogs.msdn.microsoft.com/directx/2018/03/19/announcing-microsoft-directx-raytracing/ (Accessed 2018-04-09).

[46] N. Subtil, 'NVIDIA RTX: Enabling Ray Tracing in Vulkan,' ser. NVIDIA GPU Technology Conference, 2018. [Online]. Available: http://on-demand.gputechconf.com/gtc/2018/presentation/s8521-advanced-graphics-extensions-for-vulkan.pdf (Accessed 2018-10-12).

[47] 'Metal for Accelerating Ray Tracing,' 2018. [Online]. Available: https://developer.apple.com/documentation/metalperformanceshaders/metal_for_accelerating_ray_tracing

[48] 'Real-Time Ray-Tracing Techniques for Integration into Existing Renderers,' presented at GDC 2018. [Online]. Available: https://gpuopen.com/gdc-2018-presentation-real-time-ray-tracing-techniques-integration-existing-renderers/

[49] 'Metal Shading Language Specification.' [Online]. Available: https://developer.apple.com/metal/Metal-Shading-Language-Specification.pdf

[50] 'White Paper | AMD Graphics Cores Next (GCN) Architecture.' [Online]. Available: https://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf (Accessed 2019-01-03).

[51] S. Aaltonen, 'Optimizing GPU occupancy and resource usage with large thread groups,' 2017. [Online]. Available: https://gpuopen.com/optimizing-gpu-occupancy-resource-usage-large-thread-groups/ (Accessed 2018-08-12).

[52] J. Novák, V. Havran, and C. Daschbacher, 'Path Regeneration for Interactive Path Tracing.' Eurographics Association, 2010, p. 61–64.

[53] I. Wald, 'Active Thread Compaction for GPU Path Tracing,' in *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, ser. HPG '11. New York, NY, USA: ACM, 2011, p. 51–58. doi:10.1145/2018323.2018331

[54] J. Reinders, 'Intel® AVX-512 Instructions,' 2017. [Online]. Available: https://software.intel.com/en-us/blogs/2013/avx-512-instructions (Accessed 2018-08-12).

[55] S. Baxter, 'Mergesort,' 2013. [Online]. Available: https://moderngpu.github.io/mergesort.html (Accessed 2018-05-03).

[56] T. Harada and L. Howes, 'Introduction to GPU Radix Sort,' 2011. [Online]. Available: http://www.heterogeneouscompute.org/wordpress/wp-content/uploads/2011/06/RadixSort.pdf (Accessed 2018-10-05).

[57] B. P. Welford, 'Note on a Method for Calculating Corrected Sums of Squares and Products,' *Technometrics*, vol. 4, no. 3, p. 419–420, 1962. [Online]. Available: http://www.jstor.org/stable/1266577

[58] Apple Inc., 'MPSRayIntersector Documentation,' macOS 10.14 SDK.

[59] B. Bitterli, 'Rendering resources,' 2016, https://benedikt-bitterli.me/resources/. (Accessed 2019-01-14).

[60] LuxCoreRender Project, 'Luxrender.' [Online]. Available: https://luxcorerender.org (Accessed 2018-11-03).

[61] M. E. Lee, R. A. Redner, and S. P. Uselton, 'Statistically Optimized Sampling for Distributed Ray Tracing,' in *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '85. New York, NY, USA: ACM, 1985, p. 61–68. doi:10.1145/325334.325179

[62] D. Kirk and J. Arvo, 'Unbiased Sampling Techniques for Image Synthesis,' in *Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '91. New York, NY, USA: ACM, 1991, p. 153–156. doi:10.1145/122718.122735

[63] D. P. Mitchell, 'Generating Antialiased Images at Low Sampling Densities,' *SIGGRAPH Comput. Graph.*, vol. 21, no. 4, p. 65–72, Aug. 1987.

[64] B. Karis, 'High Quality Temporal Supersampling,' in *ACM SIGGRAPH 2014: Advances in Real-Time Rendering in Games*, ser. SIGGRAPH '14. ACM, 2014. [Online]. Available: http://advances.realtimerendering.com/s2014/index.html

[65] 'Thekla Atlas.' [Online]. Available: https://github.com/Thekla/thekla_atlas (Accessed 2018-06-14).

[66] VoidStar, 'Find nth set bit in an int,' Oct 2011. [Online]. Available: https://stackoverflow.com/questions/7669057/find-nth-set-bit-in-an-int (Accessed 2018-07-04).

[67] S. G. Chang and G. S. Yovanof, 'A simple block-based lossless image compression scheme,' in *Conference Record of The Thirtieth Asilomar Conference on Signals, Systems and Computers*, vol. 1, Nov 1996, pp. 591--595 vol.1. doi:10.1109/ACSSC.1996.601093

[68] J.-H. Nah, Y.-H. Jung, W.-C. Park, and T.-D. Han, 'Efficient ray sorting for the tracing of incoherent rays,' *IEICE Electronics Express*, vol. 9, no. 9, p. 849–854, 2012.

[69] P. Christensen, A. Kensler, and C. Kilpatrick, 'Progressive Multi-Jittered Sample Sequences,' *Computer Graphics Forum*, vol. 37, no. 4, p. 21–33, 2018.

[70] A. Kensler, 'Correlated multi-jittered sampling,' *Pixar Technical Memos*, vol. 13-01, 03 2013. [Online]. Available: https://graphics.pixar.com/library/MultiJitteredSampling/paper.pdf

[71] I. Sobol', 'On the distribution of points in a cube and the approximate evaluation of integrals,' *USSR Computational Mathematics and Mathematical Physics*, vol. 7, no. 4, pp. 86 -- 112, 1967. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0041555367901449

[72] E. Heitz, 'Sampling the GGX Distribution of Visible Normals,' *Journal of Computer Graphics Techniques (JCGT)*, vol. 7, no. 4, p. 1–13, November 2018. [Online]. Available: http://jcgt.org/published/0007/04/01/

[73] A. Weidlich and A. Wilkie, 'Arbitrarily layered micro-facet surfaces,' *Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia - GRAPHITE 07*, 2007. doi:10.1145/1321261.1321292

[74] W. Jakob, E. Deon, O. Jakob, and S. Marschner, 'A comprehensive framework for rendering layered materials,' *ACM Transactions on Graphics*, vol. 33, no. 4, p. 1–14, 2014.

[75] P. Vévoda and J. Kɒivánek, 'Adaptive Direct Illumination Sampling,' in *SIGGRAPH ASIA 2016 Posters*, ser. SA '16. New York, NY, USA: ACM, 2016, p. 43:1–43:2. doi:10.1145/3005274.3005283

[76] A. C. Estevez and C. Kulla, 'Importance Sampling of Many Lights with Adaptive Tree Splitting,' in *ACM SIGGRAPH 2017 Talks*, ser. SIGGRAPH '17. New York, NY, USA: ACM, 2017, p. 33:1–33:2. doi:10.1145/3084363.3085028

[77] E. Veach and L. J. Guibas, 'Optimally Combining Sampling Techniques for Monte Carlo Rendering,' in *Proceedings of the 22Nd Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '95. New York, NY, USA: ACM, 1995, p. 419–428. doi:10.1145/218380.218498

[78] E. Persson, 'Practical Clustered Shading,' 2013, presented at SIGGRAPH 2013.

[79] O. Olsson, M. Billeter, and U. Assarsson, 'Clustered Deferred and Forward Shading,' in *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics*, ser. EGGH-HPG'12. Goslar Germany, Germany: Eurographics Association, 2012, p. 87–96. doi:10.2312/EGGH/HPG12/087-096

[80] Y. O'Donnell and M. G. Chajdas, 'Tiled light trees,' in *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ser. I3D '17. New York, NY, USA: ACM, 2017, pp. 1:1--1:7. doi:10.1145/3023368.3023376

[81] E. Heitz, S. Hill, and M. McGuire, 'Combining analytic direct illumination and stochastic shadows,' in *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, May 2018, p. 10, i3D 2018. [Online]. Available: http://www.casual-effects.com/research/Heitz2018Shadow/index.html

[82] G. J. Ward, R. D. Clear, and F. Rubinstein, 'A Ray Tracing Solution for Diffuse Interreflection,' *ACM Computer Graphics*, vol. 22, no. 4, Aug 1998. doi:ACM-0-89791-275-6/88/008/0085

[83] J. Křivánek, P. Gautron, G. Ward, O. Arikan, and H. W. Jensen, 'Practical global illumination with irradiance caching,' *ACM SIGGRAPH 2007 courses*, 2007. doi:10.1145/1281500.1281617

[84] J. Křivánek, P. Gautron, S. Pattanaik, and K. Bouatouch, 'Radiance caching for efficient global illumination computation,' in *ACM SIGGRAPH 2008 Classes*, ser. SIGGRAPH '08. New York, NY, USA: ACM, 2008, p. 75:1–75:19. doi:10.1145/1401132.1401228

[85] B. Vogl, 'Light Probes,' September 2010. [Online]. Available: http://dativ.at/lightprobes/ (Accessed 2018-07-15).

[86] T. Roughton, 'Spherical Gaussian Encoding,' 2018. [Online]. Available: http://torust.me/rendering/irradiance-caching/spherical-gaussians/2018/09/21/spherical-gaussians.html (Accessed 2018-09-21).

[87] Y. O'Donnell, 'Probulator.' [Online]. Available: https://github.com/kayru/Probulator (Accessed 2018-10-26).

[88] P. Debevec, 'Rendering Synthetic Objects into Real Scenes: Bridging Traditional and Image-based Graphics with Global Illumination and High Dynamic Range Photography,' in *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '98. New York, NY, USA: ACM, 1998, pp. 189--198. doi:10.1145/280814.280864

[89] B. Walter, S. R. Marschner, H. Li, and K. E. Torrance, 'Microfacet Models for Refraction Through Rough Surfaces,' in *Proceedings of the 18th Eurographics Conference on Rendering Techniques*, ser. EGSR'07. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2007, p. 195–206. doi:10.2312/EGWR/EGSR07/195-206

[90] K. Xu, W.-L. Sun, Z. Dong, D.-Y. Zhao, R.-D. Wu, and S.-M. Hu, 'Anisotropic Spherical Gaussians,' *ACM Transactions on Graphics*, vol. 32, no. 6, p. 209:1–209:11, 2013.

[91] S. Hill. [Online]. Available: https://mynameismjp.wordpress.com/2016/10/09/sg-series-part-3-diffuse-lighting-from-an-sg-light-source/ (Accessed 2018-08-13).

[92] J. Arvo, 'Applications of Irradiance Tensors to the Simulation of non-Lambertian Phenomena,' in *Proceedings of the 22Nd Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '95. New York, NY, USA: ACM, 1995, pp. 335--342. doi:10.1145/218380.218467

[93] L. Belcour, G. Xie, C. Hery, M. Meyer, W. Jarosz, and D. Nowrouzezahrai, 'Integrating Clipped Spherical Harmonics Expansions,' *ACM Trans. Graph.*, vol. 37, no. 2, pp. 19:1--19:12, Mar. 2018.

[94] Mathworks, 'MATLAB.' [Online]. Available: https://mathworks.com/products/matlab.html (Accessed 2018-12-18).

[95] E. Heitz, 'Understanding the masking-shadowing function in microfacet-based brdfs,' *Journal of Computer Graphics Techniques (JCGT)*, vol. 3, no. 2, p. 48–107, June 2014. [Online]. Available: http://jcgt.org/published/0003/02/03/

[96] B. Karis, 'Real Shading in Unreal Engine 4,' in *ACM SIGGRAPH 2013 Courses*, ser. SIGGRAPH '13. ACM, 2013, p. 22:1–22:8. doi:10.1145/3084363.3085028

[97] C. B. Markwardt, 'Non-linear Least-squares Fitting in IDL with MPFIT,' in *Astronomical Data Analysis Software and Systems XVIII*, ser. Astronomical Society of the Pacific Conference Series, D. A. Bohlender, D. Durand, and P. Dowler, Eds., vol. 411, Sep. 2009, p. 251.

[98] H. Dammertz, D. Sewtz, J. Hanika, and H. P. A. Lensch, 'Edge-avoiding À-Trous Wavelet Transform for Fast Global Illumination Filtering,' in *Proceedings of the Conference on High Performance Graphics*, ser. HPG '10. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2010, pp. 67--75. [Online]. Available: http://dl.acm.org/citation.cfm?id=1921479.1921491

[99] Y. O'Donnell, *FrameGraph: Extensible Rendering Architecture in Frostbite*, 2017, presented at GDC 2017. [Online]. Available: https://www.ea.com/frostbite/news/framegraph-extensible-rendering-architecture-in-frostbite

[100] Intel Corporation, 'Embree.' [Online]. Available: https://embree.github.io (Accessed 2018-04-14).

[101] M. J. Pettineo, 'SG Series Part 5: Approximating Radiance and Irradiance with Spherical Gaussians,' 2016. [Online]. Available: https://mynameismjp.wordpress.com/2016/10/09/sg-series-part-5-approximating-radiance-and-irradiance-with-sgs/ (Accessed 2018-04-10).

[102] C. Barré-Brisebois, 'A Certain Slant of Light: Past, Present and Future Challenges of Global Illumination in Games,' 2017, presented at SIGGRAPH 2017. [Online]. Available: http://openproblems.realtimerendering.com/s2017/index.html

[103] A. Silvennoinen and J. Lehtinen, 'Real-time global illumination by precomputed local reconstruction from sparse radiance probes,' *ACM Transactions on Graphics*, vol. 36, no. 6, 2017. doi:10.1145/3130800.3130852

[104] S. Ravichandran and P. J. Narayanan, 'Coherent and importance sampled LVC BDPT on the GPU,' *SIGGRAPH ASIA 2015*, 2015. doi:10.1145/2820903.2820913

[105] C. Schied, M. Salvi, A. Kaplanyan, C. Wyman, A. Patney, C. R. A. Chaitanya, J. Burgess, S. Liu, C. Dachsbacher, A. Lefohn, and et al., 'Spatiotemporal variance-guided filtering,' *Proceedings of High Performance Graphics on - HPG 17*, 2017. doi:10.1145/3105762.3105770

[106] T. Müller, M. Gross, and J. Novák, 'Practical Path Guiding for Efficient Light-Transport Simulation,' *Computer Graphics Forum*, vol. 36, no. 4, p. 91–100, 2017.

[107] E. Veach and L. J. Guibas, 'Optimally combining sampling techniques for Monte Carlo rendering,' *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques - SIGGRAPH 95*, 1995. doi:10.1145/218380.218498

[108] M. W. Mara, M. McGuire, B. Bitterli, and W. Jarosz, 'An efficient denoising algorithm for global illumination,' in *High Performance Graphics*, 2017.

[109] T. Davidovic, J. Kþivánek, M. Hašan, and P. Slusallek, 'Progressive Light Transport Simulation on the GPU: Survey and Improvements,' *ACM Trans. Graph.*, vol. 33, no. 3, p. 29:1–29:19, Jun. 2014.

[110] P.-P. Sloan. [Online]. Available: https://twitter.com/PeterPikeSloan/status/1044482721223856128 (Accessed 2018-09-25).

[111] S. Lagarde and C. de Rousiers, 'Moving Frostbite to Physically Based Rendering,' in *Proceedings of the 41st Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH 2014. Vancouver, Canada: ACM, 2014. [Online]. Available: https://www.ea.com/frostbite/news/moving-frostbite-to-pb

[112] C. Kulla, A. Conty, C. Stein, and L. Gritz, 'Sony Pictures Imageworks Arnold,' *ACM Trans. Graph.*, vol. 37, no. 3, p. 29:1–29:18, Aug. 2018.

[113] I. Georgiev, T. Ize, M. Farnsworth, R. Montoya-Vozmediano, A. King, B. V. Lommel, A. Jimenez, O. Anson, S. Ogaki, E. Johnston, A. Herubel, D. Russell, F. Servant, and M. Fajardo, 'Arnold: A Brute-Force Production Path Tracer,' *ACM Trans. Graph.*, vol. 37, no. 3, p. 32:1–32:12, Aug. 2018.

[114] P. Lecocq, A. Dufay, G. Sourimant, and J. Marvie, 'Analytic Approximations for Real-Time Area Light Shading,' *IEEE Transactions on Visualization and Computer Graphics*, vol. 23, no. 5, pp. 1428--1441, May 2017.

[115] D. Dunbar and G. Humphreys, 'A spatial data structure for fast Poisson-disk sample generation,' *ACM Trans. Graph.*, vol. 25, pp. 503--508, 07 2006.

[116] O. Olsson, E. Persson, and M. Billeter, 'Real-time Many-light Management and Shadows with Clustered Shading,' in *ACM SIGGRAPH 2015 Courses*, ser. SIGGRAPH '15. New York, NY, USA: ACM, 2015, pp. 12:1--12:398. doi:10.1145/2776880.2792712

[117] B. He, N. Govindaraju, Q. Luo, and B. Smith, 'Efficient gather and scatter operations on graphics processors,' November 2007. [Online]. Available: https://www.microsoft.com/en-us/research/publication/efficient-gather-and-scatter-operations-on-graphics-processors/

[118] Z. Lai, Q. Luo, and X. Jia, 'Revisiting Multi-pass Scatter and Gather on GPUs,' in *Proceedings of the 47th International Conference on Parallel Processing*, ser. ICPP 2018. New York, NY, USA: ACM, 2018, pp. 25:1--25:11. doi:10.1145/3225058.3225095