2018-11-01

NetLight: Cloud Baked Indirect Illumination

Nathan Andrew Zabriskie
Brigham Young University

NetLight: Cloud Baked Indirect Illumination

Nathan Andrew Zabriskie

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Brigham Young University

ABSTRACT

NetLight: Cloud Baked Indirect Illumination

Nathan Andrew Zabriskie
Department of Computer Science, BYU
Master of Science

Indirect lighting drastically increases the realism of rendered scenes but it has traditionally been very expensive to calculate. This has long precluded its use in real-time rendering applications such as video games which have mere milliseconds to respond to user input and produce a final image. As hardware power continues to increase, however, some recently developed algorithms have started to bring real-time indirect lighting closer to reality.

Of specific interest to this paper, cloud-based rendering systems add indirect lighting to real-time scenes by splitting the rendering pipeline between a server and one or more connected clients. However, thus far they have been limited to static scenes and/or require expensive precomputation steps which limits their utility in game-like environments.

In this paper we present a system capable of providing real-time indirect lighting to fully dynamic environments. This is accomplished by modifying the light gathering step in previous systems to be more resilient to changes in scene geometry and providing indirect light information in multiple forms, depending on the type of geometry being lit. We deploy it in several scenes to measure its performance, both in terms of speed and visual appeal, and show that it produces high quality images with minimum impact on the client machine.

Keywords: Global Illumination, Real-Time Rendering, Cloud Rendering

ACKNOWLEDGMENTS

I am incredibly grateful to my adviser, Parris Egbert, for being patient while I found a project I was passionate about. This thesis would never have been finished without his support, encouragement, and willingness to let me explore.

Thanks also to Seth Holladay for his advice and for helping me think of the core piece of this thesis, the lighting of dynamic objects.

And finally a huge thank you to my family for always encouraging me to never give up, especially during the times when it felt like I was up against a problem I would never solve.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The term global illumination (GI) refers to lighting algorithms that account for light that is reflected between objects in addition to light generated from direct light sources. The addition of this indirect, bounced lighting provides increased richness and realism to rendered images—as shown figure 1.1—but it has traditionally been very expensive to calculate. Offline renderers that generate frames for animated films can spend hours calculating how light propagates through a scene in order to produce a single frame. Interactive applications such as video games, however, do not have the luxury of spending that amount of time rendering a frame.

Most games run between 30-60 frames per second leaving mere milliseconds to perform game logic, physics calculations, and produce a final frame. Because of these incredibly tight timings game rendering engines must be careful with how many effects they implement.

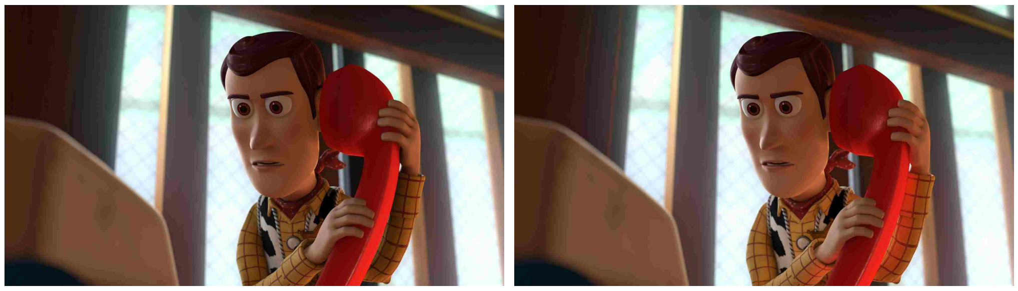


Figure 1.1: Example images from "Toy Story 3" ©Disney and Pixar showing the benefits of global illumination. The picture on the left is rendered with no GI while the image on the right includes indirect lighting. On the right, light bounces off the phone onto Woody's sleeves and hand, giving them a red glow. This subtle effect helps objects in the scene feel like they actually exist in the same space.

While the introduction of the graphics processing unit (GPU) gave game engines significantly more power to work with, they were initially quite restrictive.

Early GPUs were hard-wired to run specific rendering pipelines, making it extremely difficult to implement custom lighting algorithms using the hardware. In place of any true indirect lighting, these early systems included an ambient term that would brighten up the entire scene so that unlit areas wouldn't be completely dark. While this method was very inexpensive to calculate, it left scenes looking flat since the ambient term was uniform across the entire scene. The introduction of programmable GPUs has given graphics programmers more flexibility to experiment with alternative rendering pipelines.

There has been a great body of research developing real-time global illumination algorithms that leverage the power of modern GPUs. While great progress has been made, nearly all of these algorithms are still too slow to run in non trivial scenes on consumer-grade hardware while fitting into the 16ms-33ms window required by most games, especially with the continual rise in demand for more detailed content and effects. The rise of cloud computing, however, has opened new possibilities for pipelines that span multiple machines, offloading some or all of the rendering burden onto a remote machine.

Early remote rendering systems focused on streaming complete frames from a powerful machine to a low power display. Devices such as Valve's Steam Link and NVIDIA Shield now allow users to stream games running on their PC to a connected TV with extremely low latency. While these systems allow weaker devices to view real-time rendered content, they do not add much to the experience except portability. Recently, NVIDIA's CloudLight [7] has taken remote rendering one step further by connecting multiple rendering-capable machines.

In CloudLight and similar systems remote machines handle the more expensive GI calculations while local clients calculate direct lighting. The server sends its results to the client which injects the information into its own rendering pipeline. This greatly reduces the load on client applications so that they can remain responsive and run at high frame rates while still gaining the benefit of indirect lighting. Although these systems introduce some latency to the rendering pipeline, Crassin et al. found that the disconnect between indirect and direct lighting updates is virtually imperceptible as long as the delay is kept under half a second [7]. While remote rendering systems like Cloudlight can greatly increase lighting quality with minimal runtime impact, thus far they have only supported completely static scenes, greatly limiting their utility in game-like environments.

To build on this foundation we introduce NetLight, a remote rendering system that requires no expensive pre-baking and is capable of providing indirect lighting to static objects (as in previous systems) as well as dynamic objects that can be created, destroyed, or moved at runtime.

Chapter 2

Related Works

Most global illumination algorithms are based on some form of the rendering equation [12]:

$$L_o(x,\omega_o) = L_e(x,\omega_o) + \int_\Omega f_r(x,\omega_i,\omega_o)L_i(x,\omega_i)(\omega_i \cdot n)d\omega_i \tag{2.1}$$

where $L_o(x,\omega_o)$ is the total radiance from point x in the direction of the viewer ω_o; $L_e(x,\omega_o)$ is emitted radiance; $\int_\Omega \ldots d\omega_i$ represents an integral over Ω, the hemisphere centered around the surface normal n which contains all incoming light directions ω_i; $f_r(x,\omega_i,\omega_o)$ is the surface's BRDF which determines the proportion of light reflected from ω_i to ω_o; $L_i(x,\omega_i)$ is incoming light from direction ω_i; and $\omega_i \cdot n$ is an attenuation factor based on the angle of incoming light. Finding an analytic solution to this equation is difficult if not impossible due to the integral $\int_\Omega$ so global illumination algorithms instead typically compute a numeric approximation.

Offline rendering systems, such as those used to produce modern animated films, usually use some variation of path tracing [12]. In this algorithm, systems send millions of rays into a scene from a simulated camera. As these rays intersect scene geometry the system calculates the color of the hit object at that location and then recursively sends new rays into the scene from that point.

This family of algorithms produces high quality images but requires taking huge numbers of samples in order to denoise the resulting frame. When rendering films, this is an acceptable trade-off, but the time required to run these algorithms makes them difficult to use in real-time applications. For example, a single frame in feature animations such as Pixar's "Coco" can take over 20 hours to compute. Although path tracing has since been adapted to run on GPUs [19] and modern hardware advances have brought real-time path tracing closer to reality[1] it is still not suitable for complex scenes on typical consumer grade hardware.

The difficulty of adapting path tracing for real-time rendering has motivated researchers to develop other techniques to quickly approximate indirect lighting. Radiosity, another classic rendering technique [11], has since been adapted to run in real time on the GPU [4] but the expensive radiance-transfer calculations make it unsuitable for complicated, dynamic scenes.

Another family of algorithms called multi-light methods ([13], [9], [15], [21]) evolved as a way to approximate radiosity. In these algorithms, temporary light sources called virtual point lights (VPL) are distributed into the scene from direct light sources. During final rendering objects receive light from direct sources as well as nearby VPLs. These algorithms efficiently approximate indirect lighting in many cases. However, they all function in view space, meaning that objects outside the camera frustum do not influence lighting for visible objects. This can cause light "popping" as objects enter and exit the view. In addition, calculating shadowing for indirect light sources is expensive, requiring a ray-cast to determine visibility, again limiting viability in complicated scenes. While these algorithms produce excellent visual results, their limitations have kept them from making the jump into mainstream applications.

More recently, Crassin et al. introduced a novel method called voxel cone tracing [6] which overcomes many of the limitations of earlier real-time GI techniques. The key to this method is that it operates on a voxelized version of the scene instead of the original triangular meshes. As the voxel grid represents a pre-averaged version of the original scene, lighting at any point in the scene can be quickly approximated by taking only a few samples from the grid. This exponentially reduces the number of samples required to produce noise-free results when compared to a traditional path tracing algorithm. While this algorithm provides

[1]For a recent example see ILMxLab and NVIDIA's demo at GDC 2018.

excellent, consistent results it is still too slow to be used in many high fps applications or on low-power devices.

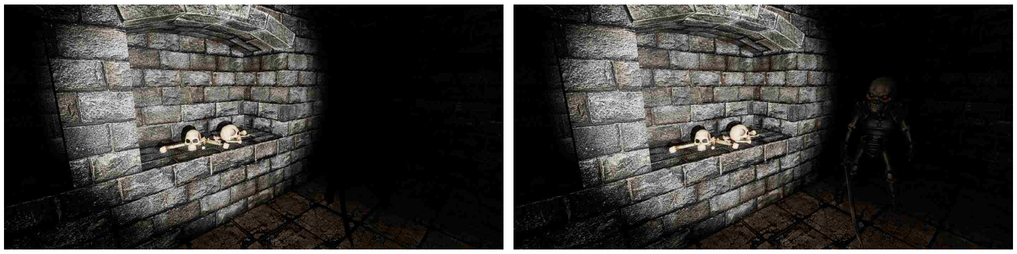
The explosion of popularity of mobile devices has generated a large interest in developing new ways to deliver content to what is essentially a portable display. Remote rendering systems such as those developed by Koller et al. [14] and Pajak et al. [10] render full frames on remote machines which are then streamed to client devices. Several studies have dealt with the viability of playing games on cloud devices (Chen et al. [2], Manzano et al. [17], Choy et al. [3]) using similar principles. While these systems do enable low-power devices to view and interact with high-quality content, they do not take advantage of the client's own capability when running on a higher power machine.

More recently NVIDIA experimented with partial remote rendering in their CloudLight system [7]. They provided three different rendering pipelines depending on the target client device, ranging from smart phones and tablets to gaming computers. On more powerful devices the server would provide only indirect lighting in some form rather than full frames. The client machine would receive this information and then add it into its own direct lighting calculations, producing a final image which included direct and indirect lighting.

Of most interest to our system is their pipeline targeting VR headsets and mid-tier devices. In this configuration their server uses an Optix GPU ray tracer[2] to generate irradiance maps which the client injects into an unmodified forward rendering pipeline. While this ray tracing step produces very high quality results, it requires an expensive pre-processing stage in order to maintain the necessary update frequency during runtime. For complex scenes they were unable to compute the necessary parameterization. Additionally, even simple scenes had to be completely static because any object that was created, destroyed, or moved at runtime would invalidate the pre-computed transfer functions.

Liu et al. removed the need to pre-calculate light transfer functions by using a mix of real-time indirect lighting algorithms in place of CloudLight's Optix ray tracer [16]. However,

[2]https://developer.nvidia.com/optix



(a) Direct + indirect lighting for static objects only (b) Direct + indirect lighting for static and dynamic objects

Figure 2.1: Games feature many characters and objects that can move at runtime which must be lit in a consistent manner. In this example the scene is lit from a single flashlight which the player can control at will. Previously developed cloud rendering systems would produce images similar to (a) where only static objects receive indirect lighting. The addition of indirect light for dynamic objects in (b) increases visual consistency and positively impacts gameplay by allowing the player to see dangers that are not directly lit by their flashlight.

their system still requires a pre-processing step to create simplified versions of every object in the scene. In addition, their system only allows users to move the camera or lights within the scene. It offers no support for geometry that is modified during runtime.

Although all of these systems produce quality images, they each have limitations which prevent them from being used in game-like scenes. In order to be effectively used in these environments, systems must be able to provide indirect lighting to dynamic objects, objects that can change position at runtime in addition to static geometry as demonstrated in figure 2.1. They must also add this functionality within the strict timing window required by modern high frames per second games. They run at 90 FPS or higher!

Thesis Statement

By modifying CloudLight's rendering pipeline to produce light probe weights in addition to irradiance maps using voxel cone tracing, we can provide high quality indirect lighting to fully dynamic scenes without any expensive precomputation step.

Chapter 3

System Overview

Our system adds interactive, dynamic indirect diffuse illumination to real-time rendering environments by separating equation 2.1 across multiple machines. The server handles the difficult approximation of $\int_{\Omega}$ and sends indirect lighting information to one or more client machines in the form of irradiance maps and spherical harmonic (SH) weights. Each client independently calculates direct lighting for the scene and samples the appropriate indirect lighting representation received from the server. By calculating and sending SH weights in addition to irradiance maps, our system is capable of providing indirect light to both static and dynamic objects. This has not been possible in previous systems which only supported static geometry.

Figure 3.1 shows an overview of the system and the following sections detail the steps used to calculate indirect lighting on NetLight's server. These steps are:

1. Voxelize the scene into a dense 3D texture. This creates a pre-filtered representation of each mesh as lit by direct lighting which helps expedite the later light-gathering steps.
2. Perform cone traces within the voxel grid to gather indirect light
3. Store indirect light for static objects in an irradiance map
4. Update the SH coefficients of each light probe in the scene to represent the current directional irradiance at that position
5. Encode the irradiance map and send it and the new SH coefficients to connected clients

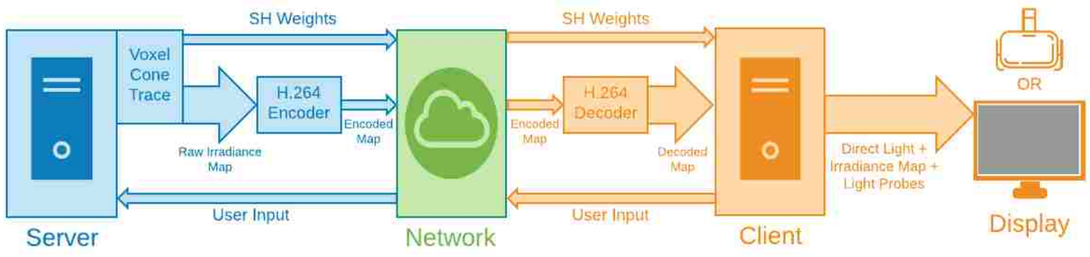


Figure 3.1: The multi-machine pipeline of the NetLight system. Indirect lighting is calculated on the server (blue) and transferred to the client (orange) across a local or wide network (green). Arrow thickness indicates relative bandwidth requirement. Lighting information is sent across the network in two forms. Spherical harmonic weights are calculated and transferred in their raw form. Irradiance maps are encoded using an H.264 encoder before transmission.

Each client receives the irradiance map and SH coefficients and injects them into a typical forward rendering pipeline. By transmitting the calculated irradiance in these forms NetLight can be integrated into existing rendering engines with little modification on the client side. All results in this paper were gathered using the Unity game engine, but NetLight could be adapted to target many real-time rendering systems which use pre-baked irradiance maps and light probes.

3.1 Voxelization

The server approximates incoming indirect light for scene objects using a modified version of the voxel cone tracing algorithm first introduced by Crassin et al. [6]. Before each lighting update, the system calculates and stores direct lighting information into a dense voxel grid. Saving the average color of objects that fall within each cell creates a pre-filtered representation of the scene. Operating on this pre-averaged grid allows the algorithm to quickly and reliably gather low-frequency indirect lighting without the need to calculate expensive intersections with the original triangle meshes.

Our system voxelizes the scene using the method described by Cozzi and Riccio [5]. The remainder of this section outlines their process which cleverly exploits the geometry shader stage and random texture read/write access to completely voxelize the scene in a single render pass.

The geometry shader first examines each triangle's normal n and uses it to determine the main scene axis $v_{\{x,y,z\}}$ which maximizes the equation

$$l_{\{x,y,z\}} = |n \cdot v_{\{x,y,z\}}| \tag{3.1}$$

It then performs an orthographic projection along the selected axis producing *voxel fragments*, 3D versions of the typical 2D fragments. Projecting along the dominant axis ensures that the resulting projection produces the maximum number of voxel fragments. This in turn helps reduce gaps in the resulting voxelization.

By setting the viewport resolution to match the voxel grid resolution (e.g. (128, 128) for a 128^3 voxel grid) these fragments can easily be mapped back into the grid and stored in the appropriate cell. During storage it is likely that multiple voxel fragments will map to the same cell. Neglecting to account for these collisions would have the effect that the last fragment to map to each cell would completely determine its color. This could cause undesirable flickering between frames. To prevent this flickering and to create a more accurate approximation of the scene we use the custom running average atomic function described in [5].

Current versions of DirectX and OpenGL do not support atomic operations on floating-point data types without extensions. However, an atomic add operation can be emulated by packing the data into a *uint* and using the *InterlockedCompareExchange* function to create a spinlock. When each thread acquires the lock it simply adds the color of its voxel fragment to the running average kept in the appropriate cell. This prevents flickering between frames and produces a better final approximation of the scene.

While the dense 3D grid used by our system occupies more memory than the sparse octree used in [6], storing the grid as a 3D texture allows for highly efficient sampling

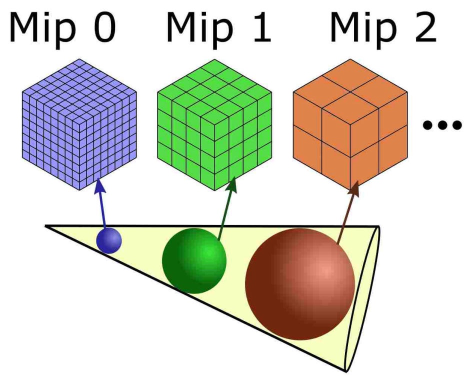


Figure 3.2: During final light gathering the system takes multiple quadralinearly interpolated samples from the voxel grid along the length of a cone. For each sample the system selects an appropriate mip level based on the current width of the cone. The samples' color and occlusion factor are accumulated along the length of the cone and returned when the maximum length is reached or the occlusion reaches a pre-set value.

during final gathering. This allows the system to avoid expensive tree traversals and take full advantage of the GPU's texture-sampling hardware, making it much more efficient in practice [18].

3.2 Light Gathering

After the scene has been voxelized, the system gathers indirect light by performing cone traces within the voxel grid. As introduced by Amanatides [1], using cones in place of infinitesimally thin rays in a ray tracing algorithm approximates the results of tracing "bundles" of many directionally coherent rays. The results are naturally anti-aliased which allow cone tracing algorithms to produce noise-free images using orders of magnitude fewer samples than would be required with traditional rays. When operating directly on scene geometry, calculating cone-primitive intersections can be highly complex and prohibitively expensive for use in real-time applications. However, the voxel cone tracing alrgorithm as introduced by Crassin et al. [6] bypasses the need to perform any intersection testing by operating on an approximation of the scene's geometry.

Our system uses a modified version of the original voxel cone tracing algorithm in which the voxel grid is stored in a mip-mapped 3D texture instead of a sparse octree. As discussed in section 3.1, this storage method sacrifices GPU memory in order to avoid expensive tree traversals during light gathering. Each mip level of the dense texture represents a pre-averaged version of the scene at increasingly coarse resolutions. Sampling a particular mip level returns an approximation of the cone-scene intersection at the corresponding world location for a cone with a diameter equal to the size of one cell. With the exception of storage method our system follows the original algorithm as described below.

To perform each cone trace our system takes multiple samples along the cone, selecting mip levels based on the current radius

$$r = l * R/L \tag{3.2}$$

where R is the radius of the cone at maximum length L and l is the current length (this process is illustrated in figure 3.2). Each sample retrieves an RGB color c_{new} and occlusion factor α_{new} which are added to a running total using the typical volumetric front-to-back accumulation formulas

$$c = \alpha_{prev} c_{prev} * (1 - \alpha_{prev}) \alpha_{new} c_{new} \tag{3.3}$$

$$\alpha = \alpha_{prev} + (1 - \alpha_{prev}) \alpha_{new} \tag{3.4}$$

Each individual cone trace terminates when the maximum cone length H is reached or α reaches a pre-determined opacity threshold at which point c is returned. Quadralinearly interpolating samples between mip levels ensures smooth transitions along the length of each cone and reduces aliasing artifacts.

By sampling the mip-mapped voxel grid in this fashion the system approximates the light received by each cone without performing any intersections with actual scene geometry. This operation is much less expensive than calculating traditional intersections and it has very predictable runtime cost, making it well suited to run in parallel on the GPU.

For any point in the scene x, sampling multiple cones in the hemisphere Ω centered around the surface normal n produces an approximation of the total irradiance from diffuse sources:

$$L_{diff}(x) = \frac{\sum_{j=0}^{C} g(x, d_j)(n \cdot d_j)}{C} \tag{3.5}$$

where $L_{diff}(x)$ is the total irradiance at point x from indirect diffuse sources, C is the number of cones, d_j is the direction of cone j selected from Ω, and g is the cone trace function. In our system L_{diff} serves as an approximation of the full integral $\int_\Omega$ when accounting only for incoming diffuse radiance, thus:

$$L_{diff}(x) \approx \int_\Omega L_i(x, \omega_i)(\omega_i \cdot n) d\omega_i \tag{3.6}$$

When L_{diff} is calculated on the same machine as final rendering it can be used to shade pixels directly. In our system this information must first be transmitted to a secondary machine which then injects it into its own rendering pipeline. To provide maximum compatibility with existing pipelines our system provides L_{diff} in two common formats: irradiance lightmaps for shading static scene elements and spherical harmonic weights for dynamic objects.

3.3 Irradiance Map Generation

Irradiance maps are simply images that contain the color and intensity of indirect lighting for objects in texture space. They have long been used in real-time rendering applications to add high quality indirect lighting with minimal runtime costs. Most systems create these maps as part of a pre-baking phase using path tracing or some other high quality global illumination algorithm. Then during runtime these textures are sampled and the results added in to direct lighting calculations that occur each frame.

While the results of this process look visually appealing, typical irradiance maps are invalidated whenever lighting conditions or scene geometry change. Because the maps typically cannot be recomputed at runtime, most lightmapped scenes feature static geometry and fixed light sources.

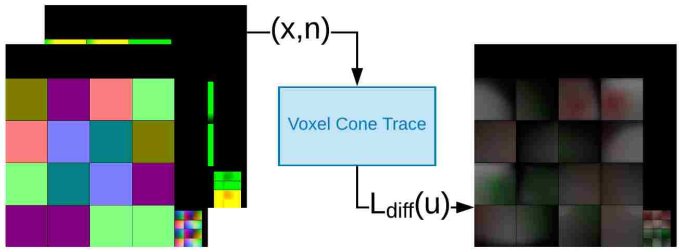


Figure 3.3: Updating irradiance maps on the server. Two buffers store world-space positions, x, and normals, n, for static objects. The retrieved values for x and n are used to perform a cone trace and the results are stored in the final irradiance map.

Previous remote rendering systems have added the ability to change lighting conditions at runtime by performing the expensive irradiance map updates on a server and sending the results to connected clients. However, both CloudLight and the system introduced by Liu et al. require expensive pre-processing steps that are invalidated if the scene's geometry changes. To achieve our goal of supporting dynamic scenes we had to find a renderer that did not rely on any calculations that might be invalidated at runtime. After weighing several options we eventually selected the voxel cone tracing algorithm as described in section 3.2. This algorithm was specifically developed with the goal of eliminating any pre-processing step and naturally supports dynamic scenes, making it an ideal candidate. Although it was originally introduced in a deferred rendering context, we have adapted it to instead produce irradiance maps as explained in the rest of this section.

At startup NetLight's server renders the world position and world normal of each static object into two texture-space buffers. Whenever the irradiance map needs to be updated the server samples these buffers, retrieves x and n, and gathers indirect light using equation 3.5. The resulting color is stored in the corresponding lumel in the final irradiance map as $L_{diff}(u)$ where u is a lightmap uv-coordinate as shown in figure 3.3. Currently the server recalculates L_{diff} in its entirety every time the server updates the irradiance map, regardless of how much the scene has changed since the last update. Once the server completes the update it sends the resulting irradiance maps to connected clients.

As each client receives a new irradiance map it simply overwrites any previous results. Whenever the client renders a frame, it samples the latest irradiance map to add indirect light to its own direct lighting calculations:

$$L_{total}(x) = L_{diff}(u) + \sum^{l} s(light_l, x) \tag{3.7}$$

where s is a typical shadow-mapping function that runs each frame to calculate if point x is in shadow for direct light source $light_l$ and performs any necessary attenuation. For static objects the server is able to provide updated irradiance maps well within the 0.5 second window required for indirect lighting to remain plausible [7]. This lengthy window, however, does not necessarily apply to dynamic objects.

Unlike static scene geometry, dynamic objects can affect their own lighting conditions by moving through the scene. For fast-moving objects, lighting conditions can change drastically within fractions of a second, quickly invalidating previous indirect lighting calculations. Thus, it is necessary that each client be able to independently adjust lighting for dynamic objects between updates from the server. To enable this capability we provide indirect lighting information in a second form: spherical harmonic weights for use in a light probe system.

3.4 Updating Light Probes

Light probes are simply points in the scene at which the system calculates the directional irradiance and stores it for later retrieval. They are often used in modern game engines as part of a mixed lighting system where dynamic objects receive direct lighting from light sources in the scene and baked indirect lighting from nearby probes.

In these systems the engine calculates directional irradiance at each probe's location and stores the result during a pre-baking phase before the application can be started. At runtime the system determines which probes should influence each object and retrieves the lighting information stored in those probes. This information is then used in combination with direct lighting to fully illuminate the object. While light probes can store lighting information in several different forms, they usually contain a series of spherical harmonic weights.

Spherical harmonics (SH) are a set of orthonormal basis functions which can be used to form compact representations of functions defined over a sphere. These basis functions are usually written as Y_l^m where l is the function's "band." Each band contains $2l+1$ polynomial functions of degree l, where $-l \leq m \leq l$.[1]

Projecting $f(s)$—a function defined over sphere S—onto the spherical harmonic basis is performed by integrating against the basis functions

$$f_l^m = \int_S f(s) y_l^m(s) ds \tag{3.8}$$

where f_l^m is the weight associated with each basis function y_l^m. These weights can later be used to reconstruct the approximation of f, $\tilde{f}$

$$\tilde{f}(s) = \sum_{i=0}^{n^2} f_i y_i(s) \tag{3.9}$$

where $i = l(l+1)+m$. By capturing directional irradiance at the position of each probe and projecting the results, the lighting environment at that position can be compactly encoded in a few weights. However, updating large numbers of weights during runtime has traditionally been prohibitively expensive.

Most systems calculate f_l^m during a pre-baking phase in which the system raytraces the scene from each probe's position and projects the results of each ray onto the SH basis functions. However, this approach requires a large number of rays to denoise the resulting irradiance function $f(s)$. Additionally, as scene complexity increases, the time required to ray trace the scene grows exponentially which excludes this technique from being used to update probes in real time.

Another approach involves generating a cubemap centered at each probe's positions and then integrating it against the SH basis functions as in [20]. While this method can

[1]For a complete explanation of how to compute the basis functions we refer the reader to Peter-Pike Sloan's presentation [22]

Num Probes	Cone Trace	Apply
32	0.00871 ms	0.02505 ms
128	0.01016 ms	0.06315 ms
512	0.00945 ms	0.20346 ms
2048	0.01157 ms	0.77979 ms

Table 3.1: Timings for generating new SH coefficients by cone tracing and applying the results to Unity's light probes.

easily be used to update each probe in real time, it involves rendering the scene six times per probe and iterating over each pixel which quickly becomes prohibitively expensive as the number of probes increases. To allow each probe to be updated during runtime, we introduce the novel method of calculating the new SH coefficients by performing cone traces in the previously computed voxel grid.

To calculate f_l^m NetLight performs a spherical cone trace at each probe's position x_p to estimate $f(s)$ and projects the result onto the SH basis functions:

$$f_l^m = \sum_{i=0}^{C} g(x_p, d_i) y_l^m(d_i) \tag{3.10}$$

where x_p is the position of each probe, C is the number of cones, and d_i is the direction of cone i. Our system calculates f_l^m for each probe in parallel using a single compute shader. By re-using the already-computed voxel grid this step can be executed with minimal runtime cost regardless of scene complexity or the number of probes in the scene as shown in table 3.1. Once the server finishes calculating the new SH weights for each probe, it sends them to connected client machines.

As each client receives the updated weights, it injects them into Unity's existing light probe system [8]. In this system the user-placed probes are organized into a tetrahedral grid. As objects move through the scene, Unity tracks which tetrahedron the object resides in.

During final rendering Unity's light probe system retrieves the weights of the probes at the vertices of the appropriate tetrahedron and interpolates between them according to the barycentric coordinates of the object within the cell. The interpolated value of f_l^m is then

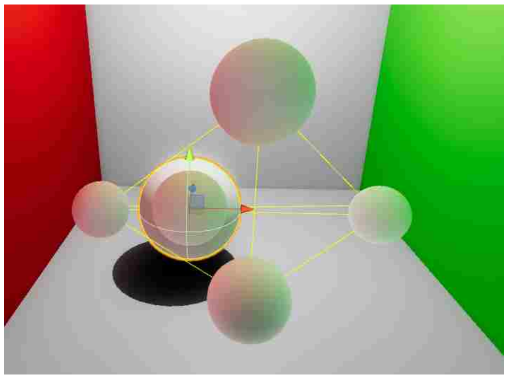


Figure 3.4: Unity's light probe system with SH weights calculated by NetLight. Although more probes exist in the scene, Unity shades the highlighted sphere using weights interpolated from the four nearest probes.

fed into equation 3.9 with s set to the surface normal n to calculate $\tilde{f}(n)$, the approximate irradiance for a surface with normal n at the object's current position. This value is then added to the light received from direct light sources when calculating the final color of the object as shown in figure 3.4.

The process described above allows the server to generate single-bounce indirect lighting and send it to connected clients for static and dynamic objects. While even single-bounce lighting adds a great deal of realism to rendered images, some interesting effects can only be achieved if light is allowed to propagate further. To add this functionality the server feeds the generated light maps and SH weights back into its own rendering pipeline. This feedback loop generates additional light bounces with a single frame delay per additional bounce. Although each light bounce b is based on the scene as it appeared in $frame_{current-b+1}$ any adverse effects are negligible.

3.5 Network Transmission

After the server has finished updating the irradiance map and light probe SH coefficients it sends them to connected client machines. As in CloudLight and the system described by Liu et al., our system first encodes the irradiance map as video using an H.264 encoder in order to reduce bandwidth. The SH values are sent in their raw form. As the client receives these updates it decodes the irradiance map and injects it and the new SH coefficients into an unmodified forward rendering pipeline as explained in the introduction to this chapter.

By splitting the full pipeline across multiple machines, the client need only calculate direct lighting each frame. This allows it to remain responsive and maintain high frame rates while still gaining the benefit of indirect lighting.

Chapter 4

Results

When measuring the utility of a remote rendering system there are several important metrics to consider. Not only must the system produce effects that are visually plausible and appealing to users, it must also give some benefit over a system that runs entirely on one machine in order to justify the added complexity.

To test our system against these metrics we deployed it in a number of scenes ranging from a simple Cornell box to a more complicated, game-like environment as summarized in table 4.1. Our experiments used one client computer equipped with an NVIDIA GeForce GTX 1070 GPU with 8GB VRAM, an Intel i5-6600K 3.5GHz CPU, and 16GB RAM. The single server machine was similarly equipped except for the GPU which was an NVIDIA Geforce GTX 1080 with 8GB VRAM. All images and data were captured from live scenes with lightmaps and light probes calculated in real time.

4.1 Visual Appearance

Our system succeeds in adding indirect lighting to a variety of environments for static and dynamic objects. Figure 4.1 shows a simple example demonstrating this basic capability. In

Scene	Triangles	Lightmap Res.	Light Probes
Cornell	4k	256x256	21
Door	11k	512x512	N/A
Tomb (VR)	518k	1024x1024	75

Table 4.1: Summary of test scenes used to gather results for section 4. Figure 4.1 shows an example image from the Cornell scene, figure 4.2 the door scene, and figure 4.3 the tomb. All tomb scene measurements were taken while outputting to an Oculus Rift.

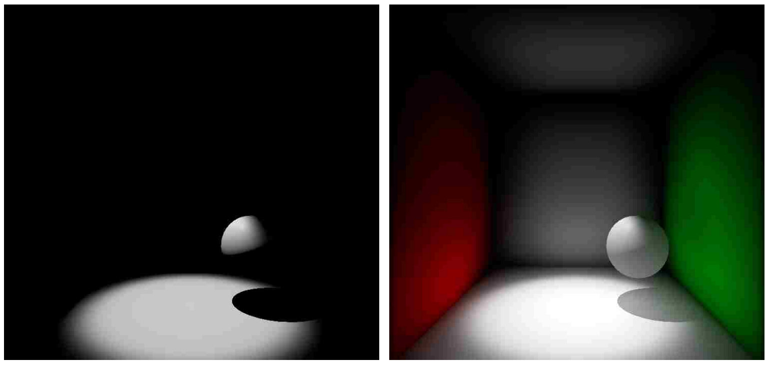

(a) Direct lighting only (b) Direct + indirect lighting

Figure 4.1: A simple scene demonstrating the system's indirect lighting capability.

this scene a dynamic sphere moves through a room lit by a single overhead light source. The system calculates lightmaps to add indirect lighting to the walls while lightprobes light the underside of the sphere.

The system's capability is further shown in figure 4.2. In this example light shines into a dark room through a narrow doorway. With single-bounce indirect lighting the details of the room's geometry become clear and a green sphere becomes visible. Once multi-bounce lighting is added, the back room takes on a green hue as a result of indirect color bleeding from the green sphere and the darkest corners show more detail.

While these simple examples clearly show the indirect lighting added by the system, the benefits become clearer in a more complicated scene. Figure 4.3 shows the Catacomb test scene in which several high-polygon, dynamic models are placed into an environment lit by a

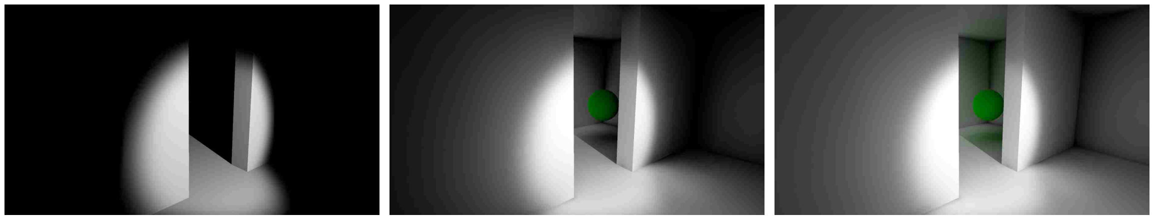
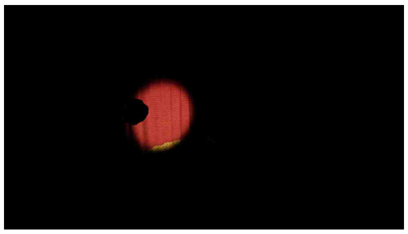
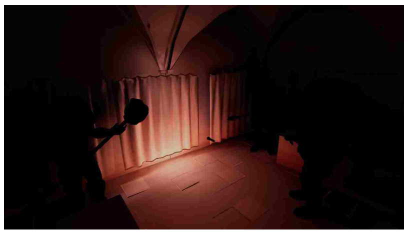
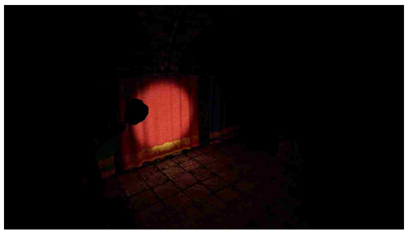
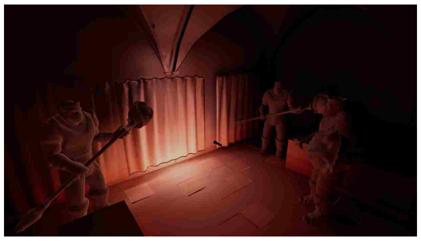

(a) The scene as lit by only direct lighting (b) Direct + single-bounce indirect light (c) Direct + multi-bounce indirect light

Figure 4.2: Single bounce lighting in (b) fills in much of the detail that is invisible with direct lighting only (a). The addition of multi-bounce lighting further fills in dark corners and adds color bleeding from indirect sources (c).


(a) Direct lighting only


(b) Static indirect lighting


(c) Direct + lightmaps


(d) Full indirect lighting

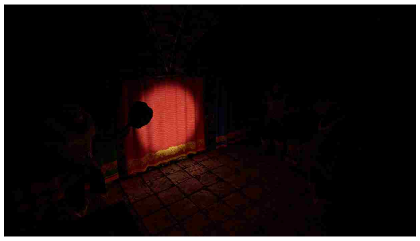

(e) Final image

Figure 4.3: An example of the indirect lighting added by our system in a game-like environment. Indirect lighting levels can be controlled independently for static and dynamic objects which use lightmaps and light probes respectively.

single flashlight. When only direct lighting is applied the room remains largely in shadow and few details are visible. As lightmaps and light probes are added the rest of the room receives indirect lighting that bounces off of the red curtain, giving the entire scene a red hue.

The use of light probes allows our system to seamlessly adjust lighting for each dynamic model as they move and animate through the scene without needing to wait for updates from the server. This capability allows our system to be used in more game-like scenarios where lighting conditions can change rapidly for animated characters. This was not possible in previous systems which focused exclusively on static geometry. In order for this new capability to be worthwhile, however, it must not overly burden the client machine.

4.2 Performance

The time required to perform each step of the indirect lighting calculations is summarized in table 4.2. It is important to note that the scene voxelization is the only step which is affected by the complexity of the scene geometry/lighting. Generating mip levels depends only on the

Scene	Voxelize	Gen. Mips	Gen. Map	Gen. Probes	Copy Probes to CPU	Total
Cornell	0.44 ms	0.038 ms	0.018 ms	0.007 ms	1.07 ms	1.573 ms
Door	0.93 ms	0.050 ms	0.024 ms	N/A	N/A	1.004 ms
Tomb (VR)	1.28 ms	0.055 ms	0.026 ms	0.0112 ms	17.01 ms	18.382 ms
Tomb	1.13ms	0.032 ms	0.023 ms	0.019 ms	9.09 ms	10.294 ms

Table 4.2: Timings for each step taken by the server whenever it updates indirect lighting. In our tests NetLight was configured to perform this update at a rate of 25Hz.

resolution of the voxel grid while the time required to generate the irradiance map and light probe coefficients depends on the lightmap resolution and number of probes.

Unfortunately, when generating light probe coefficients NetLight must copy the updated SH weights from the GPU to the CPU in order to overwrite Unity's existing light probes. In Unity this requires flushing the GPU's work queue which in our tomb scene incurred a high runtime cost, especially when targeting the Oculus Rift. If NetLight were deployed in a different engine that didn't require a GPU flush to perform this operation or allowed asynchronous reads from the GPU we anticipate that this cost would drastically decrease. As it stands with this limitation, when targeting a traditional monitor at 60 FPS our system was easily able to calculate indirect and direct lighting at a consistent framerate on a single machine.

Running the entire pipeline on one machine for the Tomb test scene and targeting an Oculus Rift (with a recommended 90Hz refresh rate) increased the average ms/frame to 14.7ms, producing 68 FPS. This caused screen tearing and other lighting artifacts which are especially noticeable and indeed uncomfortable when viewed in virtual reality. When split across two machines, however, our system was once again easily able to maintain the requisite framerate. Over a 1500 frame test, NetLight's client averaged 6.96 ms/frame

Table 4.3 summarizes the time taken to perform the additional steps required when the system runs on a separate server and client machine. As described in section 3.3 the server first encodes each lightmap before sending it to connected clients. Even with a hardware-accelerated encoder this step is expensive, as shown in table 4.3. In this configuration,

Scene	Encode Tex (Server)	Decode Tex (Client)	Apply Probes (Client)
Box	1.77 ms	0.058 ms	0.014 ms
Door	3.23 ms	0.059 ms	N/A
Tomb	6.78 ms	0.059 ms	0.060 ms

Table 4.3: Timings for additional steps introduced when our system runs across a separate server and client.

however, the server is not running the main display so the player will not notice any decrease in performance.

A background network thread on the client receives the encoded bits which it then copies to a queue on the GPU. Each frame the client's rendering thread checks for any pending frames and decodes them into the lightmap texture. In the rare circumstance where there are multiple pending frames the render thread consumes them all—since most frames in an H264 stream depend on previous frames—while keeping only the most recent frame. The H264 encoder settings we chose for our system favor rapid, low latency decoding on the client so this step is performed quickly even on larger lightmap textures as shown in table 4.3. While introducing encoding/decoding does add some computational complexity—especially on the server machine—it drastically reduces the amount of bandwidth required to run the system.

Table 4.4 summarizes the bandwidth required to transfer lighting information for each scene and the latency introduced by the network. LAN timings were collected with the server and client machines connected to the same local router through ethernet. WAN timings were

Scene	KB/s	Round Trip (LAN)	Round Trip (WAN)
Cornell	$\approx$ 256k	25.7 ms	79.09 ms
Door	$\approx$ 250k	27.2 ms	79.95 ms
Tomb	$\approx$ 473k	43.2 ms	102.03 ms

Table 4.4: Summary of required network bandwidth and latency introduced by different network configurations.

collected with the server located in Salt Lake City, Utah and client in Seattle, Washington, a distance of ≈ 700 miles.

Even over WAN our system was able to provide reactive indirect lighting updates well within the 500ms window suggested by NVIDIA's research with a maximum measured latency of 102.03ms. Additionally even our most complicated test scene required less than 0.5 MB/s of bandwidth, well within the capability of modern networks.

Chapter 5

Conclusion and Future Work

We have presented NetLight, a remote rendering system capable of adding indirect lighting for static and dynamic scenes. Its server calculates indirect lighting using voxel cone tracing which is highly resilient to increases in scene complexity and requires no pre-processing. The client application uses an out-of-the box forward rendering pipeline like those found in modern game engines with the exception that it injects results received from the server in place of pre-baked irradiance maps and light probes. This allows the system to integrate with existing applications with minimal modification.

In the future we want to investigate methods to add support for glossy reflections and other specular effects across devices. Unlike the pure diffuse lighting we currently supply, these effects depend on the viewer's position and orientation. The client must be able to move freely between updates from the server and thus it is not possible for the server to provide specular updates every frame. While the client can calculate reflections on its own using existing techniques, these algorithms are expensive to compute in real time and pre-baked solutions will not work since we have real-time GI. Ideally we would like to provide some information from the server that expedites this process.

References

[1] John Amanatides. Ray tracing with cones, 1984.

[2] Kuan-Ta Chen, Yu-Chun Chang, Po-Han Tseng, Chun-Ying Huang, and Chin-Laung Lei. Measuring the latency of cloud gaming systems. In *Proceedings of the 19th ACM International Conference on Multimedia*, MM '11, pages 1269–1272, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0616-4. doi: 10.1145/2072298.2071991. URL `http://doi.acm.org/10.1145/2072298.2071991`.

[3] Sharon Choy, Bernard Wong, Gwendal Simon, and Catherine Rosenberg. The brewing storm in cloud gaming: a measurement study on cloud to end-user latency. In *Proceedings of the 11th Annual Workshop on Network and Systems Support for Games*, page 2. IEEE Press, 2012.

[4] Greg Coombe, Mark J. Harris, and Anselmo Lastra. Radiosity on graphics hardware. In *Proceedings of Graphics Interface 2004*, GI '04, pages 161–168, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2004. Canadian Human-Computer Communications Society. ISBN 1-56881-227-2. URL `http://dl.acm.org/citation.cfm?id=1006058.1006078`.

[5] Cyril Crassin and Simon Green. Octree-based sparse voxeliztion using the GPU rasterizer. In Patrick Cozzi and Christophe Riccio, editors, *OpenGL Insights*, chapter 22. CRC Press, 2012.

[6] Cyril Crassin, Fabrice Neyret, Miguel Sainz, Simon Green, and Elmar Eisemann. Interactive indirect illumination using voxel cone tracing. In *Computer Graphics Forum*, volume 30, pages 1921–1930. Wiley Online Library, 2011.

[7] Cyril Crassin, David Luebke, Michael Mara, Morgan McGuire, Brent Oster, Peter Shirley, and Peter-Pike Sloan Chris Wyman. Cloudlight: A system for amortizing indirect lighting in real-time rendering. *Journal of Computer Graphics Techniques Vol*, 4(4):1–27, 2015.

[8] Robert Cupisz. Light probe interpolation using tetrahedral tessellations. Presentation given at the Game Developer Conference, 2012, March. URL `http://gdcvault.`

com/play/1015312/Light-Probe-Interpolation-Using-Tetrahedral. Accessed 08-23-2018.

[9] Carsten Dachsbacher and Marc Stamminger. Reflective shadow maps. In *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*, pages 203–231. ACM, 2005.

[10] Pajak Dawid, Herzog Robert, Eisemann Elmar, Myszkowski Karol, and Seidel Hans-Peter. Scalable remote rendering with depth and motion-flow augmented streaming. *Computer Graphics Forum*, 30(2):415–424. doi: 10.1111/j.1467-8659.2011.01871.x. URL https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2011.01871.x.

[11] Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Bennett Battaile. Modeling the interaction of light between diffuse surfaces. *SIGGRAPH Comput. Graph.*, 18(3):213–222, January 1984. ISSN 0097-8930. doi: 10.1145/964965.808601. URL http://doi.acm.org/10.1145/964965.808601.

[12] James T. Kajiya. The rendering equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86, pages 143–150, New York, NY, USA, 1986. ACM. ISBN 0-89791-196-2. doi: 10.1145/15922.15902.

[13] Alexander Keller. Instant radiosity. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 49–56, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co. ISBN 0-89791-896-7. doi: 10.1145/258734.258769. URL http://dx.doi.org/10.1145/258734.258769.

[14] David Koller, Michael Turitzin, Marc Levoy, Marco Tarini, Giuseppe Croccia, Paolo Cignoni, and Roberto Scopigno. Protected interactive 3d graphics via remote rendering. *ACM Trans. Graph.*, 23(3):695–703, August 2004. ISSN 0730-0301. doi: 10.1145/1015706.1015782. URL http://doi.acm.org/10.1145/1015706.1015782.

[15] Samuli Laine, Hannu Saransaari, Janne Kontkanen, Jaakko Lehtinen, and Timo Aila. Incremental instant radiosity for real-time indirect illumination. In *Proceedings of the 18th Eurographics Conference on Rendering Techniques*, EGSR'07, pages 277–286, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association. ISBN 978-3-905673-52-4. doi: 10.2312/EGWR/EGSR07/277-286. URL http://dx.doi.org/10.2312/EGWR/EGSR07/277-286.

[16] Chang Liu, Jinyuan Jia, Qian Zhang, and Lei Zhao. Lightweight websim rendering framework based on cloud-baking. In *Proceedings of the 2017 ACM SIGSIM Conference*

on Principles of Advanced Discrete Simulation, SIGSIM-PADS '17, pages 221–229, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4489-0. doi: 10.1145/3064911.3064933. URL `http://doi.acm.org/10.1145/3064911.3064933`.

[17] Marc Manzano, Jose Alberto Hernandez, M Uruenna, and Eusebi Calle. An empirical study of cloud gaming. In *Proceedings of the 11th Annual Workshop on Network and Systems Support for Games*, page 17. IEEE Press, 2012.

[18] James McLaren. The technology of the tomorrow children. Presentation given at the Game Developer Conference, 2015, March. URL `https://gdcvault.com/play/1022428/The-Technology-of-The-Tomorrow`. Accessed: 06-21-2017.

[19] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. *ACM Trans. Graph.*, 21(3):703–712, July 2002. ISSN 0730-0301. doi: 10.1145/566654.566640. URL `http://doi.acm.org/10.1145/566654.566640`.

[20] Ravi Ramamoorthi and Pat Hanrahan. An efficient representation for irradiance environment maps. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, pages 497–500, New York, NY, USA, 2001. ACM. ISBN 1-58113-374-X. doi: 10.1145/383259.383317. URL `http://doi.acm.org/10.1145/383259.383317`.

[21] Tobias Ritschel, Thorsten Grosch, Min H Kim, Hans-Peter Seidel, Carsten Dachsbacher, and Jan Kautz. Imperfect shadow maps for efficient computation of indirect illumination. *ACM Transactions on Graphics (TOG)*, 27(5):129, 2008.

[22] Peter-Pike Sloan. Stupid spherical harmonics (sh) tricks. Companion paper to a presentation given at the Game Developers Conference,, 2008. URL `http://www.ppsloan.org/publications/StupidSH36.pdf`. Accessed 06-08-2017.