Global Illumination Optimization: Integrating Interleaved Sampling into Reflective Shadow Maps and Splatting Indirect Illumination

BY
Nicholas Andrew Hahn
B.S. Computer Science, State University of New York at Buffalo, 2008

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science
in the graduate studies program
of DigiPen Institute Of Technology
Redmond, Washington
United States of America

Summer
2012

Thesis Advisor: Dr. Gary Herron

DIGIPEN INSTITUTE OF TECHNOLOGY

GRADUATE STUDY PROGRAM
DEFENSE OF THESIS

THE UNDERSIGNED VERIFY THAT THE FINAL ORAL DEFENSE OF THE MASTER OF SCIENCE THESIS OF NICHOLAS HAHN

HAS BEEN SUCCESSFULLY COMPLETED ON 07/26/2012

TITLE OF THESIS: GLOBAL ILLUMINATION OPTIMIZATION: INTEGRATING INTERLEAVED SAMPLING INTO REFLECTIVE SHADOW MAPS AND SPLATTING INDIRECT ILLUMINATION

MAJOR FILED OF STUDY: COMPUTER SCIENCE.

COMMITTEE:

Gary Herron, Chair

Xin Li

Pushpak Karnick

Antonie Boerkoel

APPROVED:

date

Graduate Program Director
Dean of Faculty

date

Department Chair of Computer Science

date

President

The material presented within this document does not necessarily reflect the opinion of the Committee, the Graduate Study Program, or DigiPen Institute of Technology.

INSTITUTE OF DIGIPEN INSTITUTE OF TECHNOLOGY

PROGRAM OF MASTER'S DEGREE

THESIS APPROVAL

DATE: *07/28/2012*

BASED ON THE CANDIDATE'S SUCCESSFUL ORAL DEFENSE, IT IS RECOMMENDED THAT THE THESIS PREPARED BY

Nicholas Hahn

ENTITLED

Global Illumination Optimization: Integrating Interleaved Sampling into Reflective Shadow Maps and Splatting Indirect Illumination

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE FROM THE PROGRAM OF MASTER'S DEGREE AT DIGIPEN INSTITUTE OF TECHNOLOGY.

Thesis Advisory Committee Chair

Director of Graduate Study Program

Dean of Faculty

Contents

Acknowledgements

I would like to thank…
My Mom, Dad, and sister: for never losing faith in me.
Ted and Tim: for motivating me.
Mary and Emily: for all their hospitality.
Gary Herron: for his guidance along the way.
Everyone who told me this was a bad idea: for giving me the drive to prove they were wrong.

Abstract

One of the main fields of study in computer graphics has been the efficient calculation of lighting and shadows for computer-generated images. Though lighting in the real world involves many complex phenomena such as soft shadows, multiple bounces of light, and caustics it was originally the case that only direct lighting and shadows could be calculated at interactive frame rates in computer environments. However, after years of research and improved graphics hardware design, it has now become possible to calculate these complex phenomena (known collectively as Global Illumination) in real-time.

This thesis has two main parts. In the first half, many of the past and current techniques for calculating indirect illumination are presented. Some of these techniques are possible in real-time, and some are only intended for offline rendering. In the second half, two of these real-time techniques, Reflective Shadow Maps and Splatting Indirect Illumination, are examined, and the results of trying to optimize them via another technique known as Interleaved Sampling are presented.

1. Introduction – Global Illumination

Modern research in computer graphics is a constantly changing and advancing field. From lighting models to post-processing effects to scene management algorithms, researchers are constantly discovering new and improved ways to render more objects and make them look better in the process. One of the most popular areas of research lately has been the field of Global Illumination. Global Illumination is an overarching term that encompasses myriad techniques focused around simulating the interaction of light with its environment and includes such techniques as sky lighting, indirect illumination, caustics, ambient occlusion and shadowing, reflection, refraction, transmission, as well as several other phenomena. The inclusion of all of these phenomena in video game rendering increases the reality of the scene being rendered, produces pleasing images for players to look at, and helps immerse the player in the game world.

All of the techniques mentioned above are attempts at approximating different aspects of the Rendering Equation that was first formulated by Jim Kajiya [Kajiya86]. Solving the Rendering Equation is a computationally expensive process that involves integrating over an entire hemisphere's worth of space at every point on every visible surface, as well as determining visibility between every arbitrary combination of surfaces. Unlike real-time applications, ray tracers have the luxury of unlimited computation time and are often able to approach closer solutions to the rendering equation than that of their real-time brethren. This thesis will focus on existing attempts to model indirect illumination (i.e. "light bleeding", "multiple light bounces", or "diffuse interreflections") in real-time. The first step on the path to understanding indirect illumination techniques in real-time applications is to take a look at the rendering equation, how it relates to GI phenomena, and why it is so difficult to solve in real-time.

2. The Rendering Equation

The Rendering Equation as defined by Jim Kajiya is as follows:

$$I(x, x') = g(x, x') \left[\epsilon(x, x') + \int_S \rho(x, x', x'')\, I(x', x'') dx'' \right]$$

Where:

$I(x, x')$ is the intensity of light passing from point x' to the point x.

$g(x, x')$ is a "geometry" term. This is either 0 or 1 depending on whether point x and x' can "see" each other. This is most accurately determined via a ray-cast from x to x'.

$\epsilon(x, x')$ is the amount of light emitted from x' that arrives at x. It is analogous to the common "emissive" term of many lighting models.

$\rho(x, x', x'')$ is the intensity of light that leaves x'' and arrives at x after reflecting off of a patch of surface at x'. This is often calculated as the Bi-Directional Reflectance Distribution Function (BRDF) of the surface at x'.

S is the union of all surfaces in the environment, including a global background surface which can be thought of as the sky in a normal scene.

x, x', and x'' are points in the environment. They can be surfaces, light sources, or the viewing camera.

Many of the phenomena that are usually considered global illumination techniques naturally occur as a result of calculating the Rendering Equation. This particular representation of the Rendering Equation is integrated over a set of surfaces, but it is possible to represent the Rendering Equation as an integral over a hemisphere of directions centered around the surface normal at point x. A solution to the Rendering Equation would provide a way to account for all of the light present in a scene, and if

calculated at every representable point in a scene would provide the most accurate lighting solution for our computer graphics scene.

Unfortunately, there are two components of the equation that are prohibitively expensive to calculate in real-time. These are the geometry term $g(x, x')$ and the recursive lighting term $I(x', x'')$. Following is a brief discussion of what makes these two terms so difficult to compute.

Ray tracing would be a reliable way to compute the geometry term between two points. However, as the complexity of a scene increases, the computational cost of performing a ray cast at every visible point becomes prohibitively expensive. To that end, techniques that take advantage of the strengths of graphics hardware and rasterization, such as shadow mapping, have been developed to help compute visibility between light sources and geometry in real time. Although it is more efficient than a ray cast, rendering a shadow map at every point quickly becomes too expensive to perform in real time. Certain techniques that compute visibility hierarchically have been developed. Some techniques, such as radiosity, precompute visibility in an offline step in an attempt to speed up the runtime computation of lighting. However, these precomputations can only account for static geometry and would not be correct for dynamic objects in a scene. The need to calculate the geometry term between every pair of points in a scene makes solving the rendering equation very expensive.

The other term in question is the recursive application of the Rendering Equation. It is necessary to take into account an entire hemisphere's worth of lighting information when determining the lighting at a pixel. This quantity, however, includes light that has bounced off of other surfaces, which received their light from other surfaces etc. etc. Attempting to recursively evaluate this term at every point in a scene can quickly become too expensive to perform in real-time. Many global illumination algorithms that are meant to be run in real-time work around this problem by only simulating direct lighting, and one or two bounces of indirect lighting.

3. Illumination as approximation (Previous Work)

The previous section highlighted why attempting to solve the Rendering Equation to produce a realistic lighting solution is prohibitively expensive for a real-time application. As a compromise, real-time applications are forced to resort to simulating different Global Illumination phenomena separately by approximating certain aspects of light transport in a scene, or through completely arbitrary non-physically based methods that produce good looking results regardless. These real-time applications typically have a separate solution for ambient occlusion, caustics, indirect illumination, soft shadows, reflections, and sky lighting. Oftentimes the solutions are completely orthogonal to one another. As this thesis is not a survey of the entire field of global illumination, only a subset of existing global illumination algorithms that relate to indirect illumination will be discussed.

The set of indirect illumination algorithms can be divided into a few higher level categories. [RDGK11] Particularly, these are Finite Element solutions, Photon Mapping solutions, and Instant Radiosity-based solutions. There are also a few techniques that do not fit into any of these categories.

3.1 Finite Element Solutions

A Finite Element Solution, or Radiosity, is an attempt to break up all of the surfaces and light sources in the scene up into smaller surface elements, called patches, and compute the lighting contribution from every patch to every other patch in the scene. This is often done in a pre-computation step, but performing this light propagation dynamically at run-time has also been attempted.

3.1.1 Radiosity

The first paper describing Radiosity [GTGB84] was based on the principles of radiative heat transfer. The paper starts with the assumption that all surfaces in a scene are ideal diffuse (Lambertian) surfaces, and that they reflect light in a uniform manner in all directions (unlike specular surfaces). Using these assumptions, it develops the concepts

of enclosures and form factors to help formulate a method of calculating the amount of light that will travel from one surface to every other surface in the scene.

An *enclosure* is defined as "a set of surfaces that completely define the illuminating environment." These surfaces can be literal, such as walls and other objects, or they can be theoretical, such as light sources. Every surface in the enclosure has material properties, such as emissive light intensity and diffuse color, defined for it. To achieve greater variation in lighting, a single literal object is often divided into smaller patches and each of those patches is added to the enclosure. This is because lighting is only calculated on a per patch basis and will be uniform across the entire face of a patch. Patches can be either emitters or reflectors, or both.

A Form Factor is defined as "the fraction of the radiant light energy leaving one particular surface which strikes a second surface." In other words, for any two surfaces, A and B, the form factor from A to B is how much of the light emanating from A that hits B, and likewise there is a form factor from B to A.

The Radiosity of a surface is the integral of all energy leaving that surface. The energy that leaves the surface can be one of two different kinds. It can be light that the surface is emitting on its own, or it can be the product of light coming into the surface, and reflecting back off of it. Although these two types of radiance are presented as two distinct quantities, they are actually identical in nature and as such it is sufficient to add them together when determining how much light is leaving a surface.

With these definitions, we can define the following equations:

$$B_j = E_j + p_j H_j$$

Where

B_j = the Radiosity of surface j. It is the total rate at which radiant energy leaves the surface in terms of energy per unit time and per unit area ($watts/meter^2$)

E_j = the rate of direct energy emission from surface j per unit time and per unit area ($watts/meter^2$)

p_j = the reflectivity of a surface j. It represents the fraction of incident light which is reflected back into the hemispherical space.

H_j = the incident radiant energy arriving at surface j per unit time and per unit area. ($watts/meter^2$)

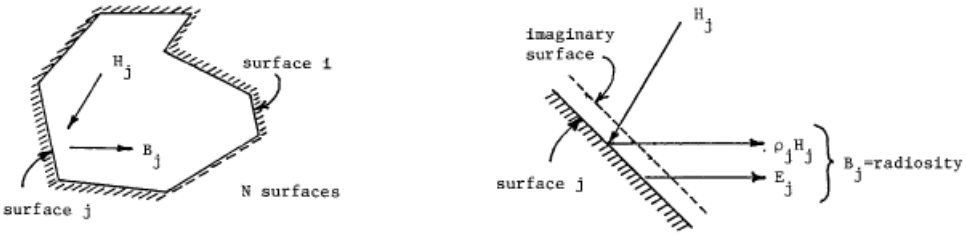
E_j and p_j are defined for each patch when the environment is created, so the only unknown in the above equation for radiosity is H_j. We can calculate H_j as follows:

$$H_j = \sum_{i=1}^{N} B_i F_{ij}$$

Where

B_i = the radiosity of surface i. ($watts/meter^2$)

F_{ij} = the form factor and represents the fraction of radiant energy leaving surface *i* and impinging on surface *j.*



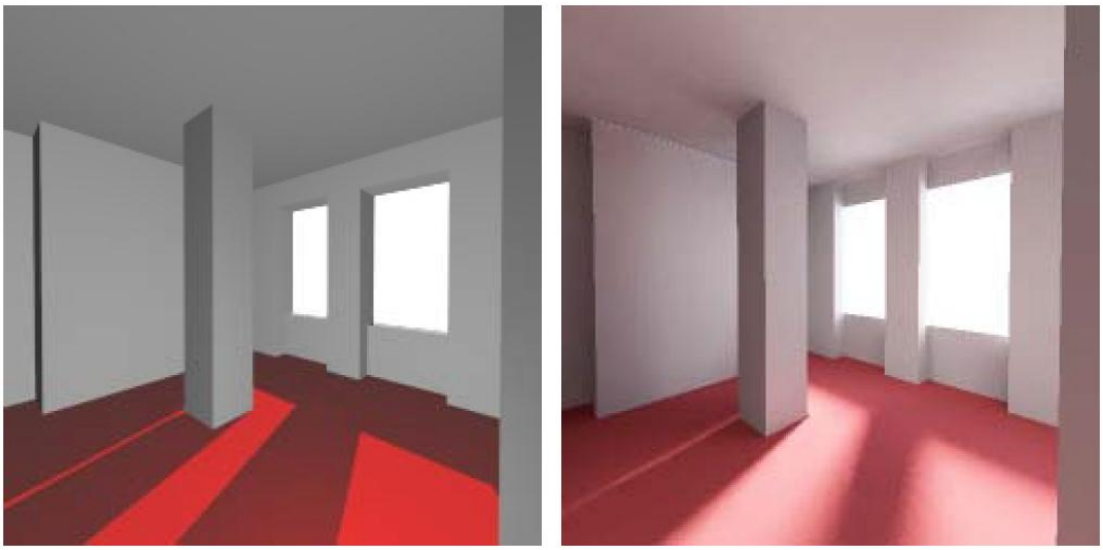
On the left is an example enclosure with N surfaces. On the right is an example of the quantities involved with calculating the Radiosity of a surface. (Image taken from [GTGB84])

We can see that this means that the incoming radiance at a patch *j* is the sum over all surfaces (including the surface *j*, it can be self-illuminating) of the product between the Radiosity of surface *i* times the form factor between surfaces *i* and *j.* We can use this information to create a system of equations to solve for the Radiosity of each patch:

$$B_j = E_j + p_j \sum_{i=1}^{N} B_i F_{ij} \ \ for\ j \ \in [1, N]$$

The only piece of the puzzle that has yet to be explained is the calculation of the form factors. This is done in an offline process which takes into consideration the differential area of each surface, the angle between each surface and their normals, and the distance between the two of them. Once the form factors are known, the above system of equations can be solved by any solver, and the radiosity of each patch is then known and stored for use at runtime. All of the work up to this point is done offline in a preprocessing step. For any reasonably complex scene, the above system of equations becomes prohibitively expensive to calculate in realtime. However, once the equations have been solved the result is a color for each patch in our scene. Once we have these colors, we can use them directly at runtime to color our scene. Since patches have a uniform color across their entire face, a form of linear smoothing can be used to produce smooth shading.

Radiosity can produce fairly plausible looking results. Effects such as indirect illumination and soft shadows can be produced for scenes that are well subdivided. The results are view-independent since diffuse light is reflected evenly in every direction. The original Radiosity technique does not allow for the calculation of specular reflections, as those are view independent and would require recalculation every frame. Some efforts have been made to add specular reflection to Radiosity, but these efforts only amounted to integrating ray tracing into radiosity, as well as introducing some new form factors into the equation. [WCG87]



The image on the left was rendered by only taking direct lighting into account. The image on the right is the result of lighting via radiosity. (Image taken from [Aliaga10])

3.1.2 Implicit Visibility and Antiradiance

Another interesting finite element solution that is similar to Radiosity but works in real-time is “Implicit Visibility and Antiradiance for Interactive Global Illumination” [DSDD07] This method recognizes that the most expensive part of the lighting equation is determining visibility and makes efforts to take visibility into account implicitly without having to explicitly calculate it.

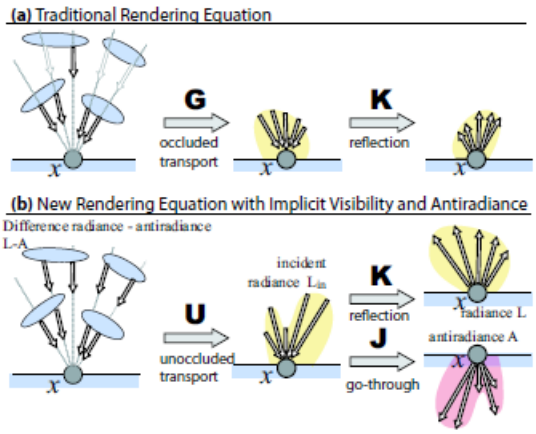
The idea behind Implicit Visibility and Antiradiance is that while a surface might reflect visible light in the direction of its normal, it can also be though to transmit an equal amount of “antiradiance” in the opposite direction. As an example, consider a scene with three rectangular boxes stacked one in front of the other like dominoes. If a light source is in front of the first domino, its light should only hit the first domino. The theory behind antiradiance is that light will get propagated unhindered to each of the three walls, but each wall will in turn propagate an equal quantity of antiradiance to the walls behind it. The final exitant radiance of a surface is then the difference of the incident radiance and the incident antiradiance. By alternately propagating radiance and antiradiance through a scene, a point of convergence can be reached where all of the surfaces that should be receiving light do, and the ones that shouldn’t, don’t.

The paper describes several “operators” used to describe how light is propagated through a scene. The operator G, when applied to light, will only affect surfaces that are visible from the light source. G causes light to only strike visible objects. J, on the other hand, is an operator that causes light to pass through a surface, as if it were a completely transparent object. U is the “unoccluded” operator, which propagates light to all surfaces in the direction of the light, regardless of whether there is an object between the source and destination. These operators can be applied in any sequence. As an example consider the sequence JGL (L is light). First G is applied to L, which results in light being propagated to surfaces directly visible to the light source. Then, J is applied to the result of GL, resulting in all of the light that strikes objects directly to be

propagated as though the surfaces were transparent. Using these operators, the authors define the terms L and A, for Light and Antiradiance.

$$L = E + KU(L - A)$$

$$A = JU(L - A)$$



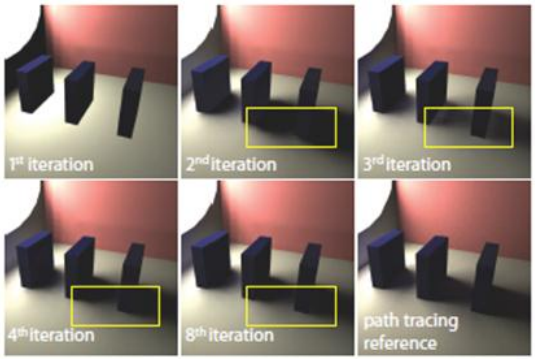
An example of the various operators. On the top is how light works in the traditional Rendering Equation. On the bottom we see how the operators can be applied to calculate Radiance and Antiradiance (Images taken from [DSDD07]).

To explain these equations, L and A propagate quantities equal to the current amount of incident radiance (L) minus the amount of incident antiradiance (A). So for a single surface, outgoing Radiance (L) is computed via the incoming light (L-A) from the last iteration without concern for visibility (U). This incoming light is thrown into a BRDF (K), and added to any naturally emitted light of the surface (E). This L will be propagated to every other object in the scene, without taking visibility (U) into account. Conversely, the object will also propagate out its antiradiance (A) based on all of the incoming light (L-A) from the last iteration without concern for visibility (U). However, as this is modified by J, this quantity is propagated behind the surface, and only hits objects behind it. By applying several iteration of L and A, the lighting in the scene eventually converges to a stable solution.

3.1.2.1 Types of Propagation

The paper makes note of two different iteration methods for propagating L and A: symmetric and asymmetric. In asymmetric iteration, one pass of L will be performed,

followed by several passes of A. These passes of A will eventually converge, after which another pass of L can be performed. This method is proven to converge, but takes much longer than symmetric iteration. In symmetric iteration, L and A are alternated, one pass of each at a time. When the amount of total iterations to be performed is rather small, symmetric iteration will actually outperform asymmetric iteration in terms of deviation error. However, asymmetric iteration has the potential to be more accurate in the end. The authors point out that after about 10 iterations of symmetric iteration, there is hardly any noticeable visible discrepancies within the scene, so it is perfectly acceptable to use symmetric iteration.



An example of how alternating radiance/antiradiance propagation can produce more accurate results as more iterations are performed. (image taken from [DSDD07])

3.1.2.2 Finite Elements

Like radiosity, antiradiance also works on many finite elements. Although antiradiance removes the need for the expensive visibility computation, if a scene has many finite elements the $O(n^2)$ element-to-element nature common to finite element methods is too computationally expensive to perform in real time. To try and speed up this process, the algorithm divides the finite elements into directional bins based on the solid angle subtended by each element's surface normal. The set of solid angles is divided into n discrete bins, and each element is placed into the appropriate bin based on its surface normal. When it comes time to propagate L and A, it is only necessary to propagate values to objects in bins that are in the appropriate hemisphere relative to the source bin. The propagation is split into two passes, a global pass and a local pass. In the global pass, exitant radiance (L-A) is propagated from sending to receiving patches

based solely on directional properties of the two patches. In the local pass, the values L and A are updated for the next iteration of the global pass. In the asymmetric approach, one iteration consists of several alternating global and local passes, but after the first local pass only antiradiance values get updated.

To further increase the efficiency of this method, hierarchical methods were applied. Directional bins can be collected into larger bins and compared against other large bins to more quickly pare down which elements need to interact with which elements. When light is propagated in a hierarchy during the global pass, the incident radiance gets pushed down to all of the leaf nodes. The local pass is performed at all leaf nodes, and the results pushed back up to the top level of the hierarchy, where the value of a node in the hierarchy is just the sum of its immediate children.

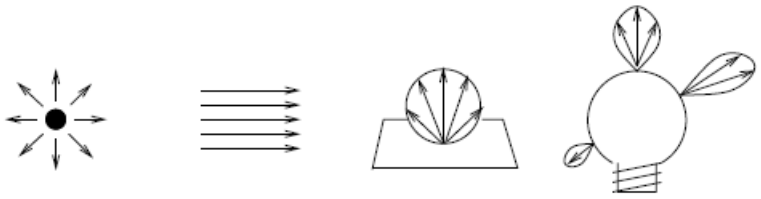
The results of this technique produce quite attractive results, but at the time of publishing, it was achieving results of 9 frames per second in a scene with 5600 triangles at 4 iterations per frame.

3.2 Photon Mapping solutions

The term Photon Mapping describes a class of techniques that work in two phases. In the first phase, a (potentially large) number of virtual photons are traced from each light source out into the scene. The process of tracing a photon involves determining intersection of that photon with the geometry in the scene, similar to how ray tracing works. Once all of the photons in a scene have been traced, the second phase, or the gather phase, occurs. In this stage, for every surface element or pixel that needs to be rendered, the N closest photons to that point are determined, and those photons are used to light the surface point that is being shaded. Under certain conditions photon mapping can be real-time, and in other situations it is only an offline process.

3.2.1 Classic Photon Mapping

The first implementation of photon mapping was described by Henrik Wann Jensen. [Jensen96] In its original form, it was not a real-time algorithm, but rather was an extension to traditional ray tracing. When talking about the photon mapping algorithm, the first thing requiring definition is the photon emission phase. As mentioned earlier, each light source in the scene emits photons. Each light source will emit photons in different directions with varying densities based on the shape and nature of the light source. A photon is defined as both a direction and a power value. The power value is drawn from the power ("wattage") of the light source, and is distributed evenly among all of the emitted photons. Ideally, photons should be emitted as uniformly as possible while following the rules of the light's shape and size. All lights in the scene will emit photons, and more powerful lights will emit a greater amount of photons to help ensure that all photons in the scene are relatively equal in power.



An example of photon emission from different types of light sources. From left to right: Point Light, Directional Light, Square Light, General Light. (Image taken from [Jensen00])

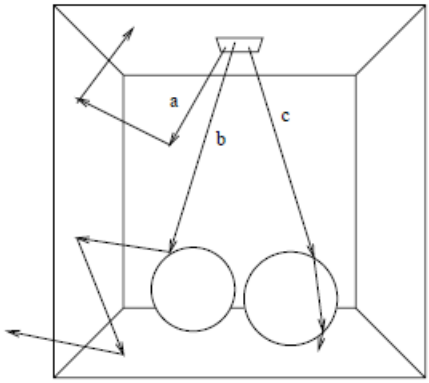
3.2.1.1 Projection Maps

To speed up the emission of photons and ensure that they hit the parts of the scene that they need to, structures known as "projection maps" are used. A projection map is a map of the geometry as seen from the light source. It is implemented as a grid where each cell holds a value indicating whether geometry exists in the direction associated with that cell. For a point light the projection map could be a spherical grid, and for a directional light it could be a 2D grid. The only condition is that any point in the scene that is visible from the point of view of the light can be mapped to a cell. It is not necessary to be completely accurate with the projection. All that is required is a conservative estimate of where the scene's geometry is. To this end a bounding volume representation of the scene can be used when creating the projection map. The projection map can be used by either emitting a photon into each cell one by one using a random direction from the set of all directions subtended by that cell until the light source has emitted enough photons, or by randomly picking among the cells that have geometry and firing a photon in that direction. Regardless of how the projection map is used, it is necessary to scale the power of each photon by the percentage of cells in the map that contain geometry. This is because technically all cells in the photon map should be receiving photons, but those photons are instead being distributed to only the cells that contain geometry.

3.2.1.2 Photon Interactions

When a photon is emitted, a ray trace occurs. Upon intersecting with an object, the photon can do one of three things. It can be reflected, transmitted, or absorbed. Which of these ends up occurring is chosen based on a Russian Roulette algorithm. Each surface has two coefficients, a diffuse reflection coefficient (*d*), and a specular reflection

coefficient (*s*) where $d + s \leq 1$. At each intersection, a uniform random variable with value in the range [0,1] is generated, and based on that random variable the photon can either be reflected or absorbed. If the random variable is in the range [0, *d*], it is reflected diffusely. If it is in the range (*d*, *s* + *d*] it is specularly reflected, and if it is in the range (*s* + *d*, 1] it is absorbed. The use of Russian Roulette prevents the need to modify the power of the reflected photons.



An example of tracing photons in a scene. Path a) shows two diffuse reflections followed by absorption, path b) shows a specular reflection followed by two diffuse reflections, and path c) shows two specular transmissions followed by absorption. (Image taken from [Jensen00])

The alternative to Russian Roulette in this case would be to generate both a diffuse and a specular photon, and scale the energy of each photons accordingly. However, this can quickly become unmanageable, as this method drastically increases the amount of photons that are used. Additionally, scaling the power values of all of those extra photons at each interaction would produce photons with drastically different power levels, which is undesirable.

Photons are only stored in a photon map when they interact with diffuse surfaces. It is fairly meaningless to track photon interactions with specular surfaces, as the probability of a photon striking at the mirrored view direction of the scene camera (where specular light would need to come from to be visible) is zero. Instead, specular reflections are calculated using traditional ray tracing. All diffuse interactions are stored in the "global photon map". Photons are also stored when they are absorbed by diffuse surfaces. The structure of a photon map is fairly simple. For each photon it stores a position, a power

value, an incident direction, and a special flag related to the kd-tree structure that the photons are being stored in.

If the scene contains any participating media (fog, mist), photon interactions with the participating media can be stored in a *volume photon map* which is separate from the global photon map. When a photon is found to be intersecting with a volume, ray marching is used along with Russian roulette to determine whether or not the light is scattered by the media or not. Volumes of uniform density are easier to calculate, but it is possible to compute interactions for variable density volumes.

3.2.1.3 Three Photon Maps

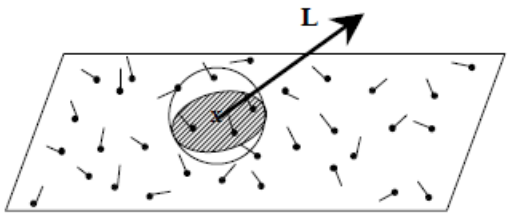
In addition to the Global Photon Map and the Volume Photon Map that have already been discussed, the original photon mapping algorithm also uses a third map called the *Caustic Photon Map*. The caustic map holds photons that have been through at least one specular reflection before hitting a diffuse surface. The caustic map contains a much denser collection of photons that are specifically fired at specular surfaces, and the global map contains photons from diffuse surfaces, which are generated with a much coarser distribution. Photon maps are stored as balanced KD-Trees to help accelerate photon lookup in the rendering phase.

An example visualization of photons stored in the Photon Map. (Image taken from [Jensen00])

3.2.1.4 Estimating Radiance

Once all of the photons are in a balanced KD-tree, the final scene can be rendered. When a pixel is to be rendered with via the ray tracer, the n-closest photons in the

photon map to the point being rendering are found. The n-closest photons to point x can be thought to be those that would be enclosed by a sphere at x that is expanded until it encloses enough photons. The resultant radiance is dependent on the sum of the contributions of each photon and the area of the sphere that encloses the *n* photons. As a 3d shape is being used to enclose photons on a surface that is flat, it is likely that error in the amount of radiance generated along seams and corners of walls will be introduced. This is because photons that are not actually on the surface that we are interested in will be enclosed by our 3d shapes. To help remedy this, different shapes can be used to enclose the n photons. A more ideal shape might be a disc, as it would be flat like the underlying surface. It is also possible to use a cylinder, but even with these more accurate shapes, it is likely that error will still be generated and filtering will be necessary. A few filters that are mentioned as being useful are cone filters, Gaussian filters, and differential checking-based filters. The goal of these filters is to apply more weight to the photons that are closer to the point that we are trying to estimate radiance at, thereby reducing a characteristic blurriness that comes from the photon sampling. The method used for searching the KD-tree for neighboring photons is a nearest-neighbor search algorithm. Photons are searched for in the kd-tree in nodes starting with the one that contains the point to be shaded, and a maximum search distance is used to help reject grid spaces that are further away than we care about.



An example of gathering the nearest photons in the photon map. (Image taken from [Jensen00])

3.2.1.5 Rendering

With the method of searching the photon map established, it is possible to perform final scene rendering. The primary method for rendering is distributed ray tracing. When trying to determine the incoming light at a point Monte Carlo integration is often used,

but with a photon map to represent irradiance in the scene the process becomes much easier. Direct illumination, specular reflection, caustics, and multiple diffuse reflections are handled as four different cases:

Direct illumination can be computed using normal ray tracing methods, but enhancements can be made with the Photon Map by incorporating the concept of *shadow photons*, which don't get explained very well. The accurate calculation here uses shadow photons, shadow rays, and light source evaluations to determine the incoming light. The approximate evaluation merely uses the values stored in the photon map as the incoming radiance. Direct illumination is an important part of any image's lighting solution so it is advisable to use a more accurate calculation rather than an approximate calculation.

The photon map is not used in the calculation of the specular/glossy reflections since those reflections are directionally dependent and the photon map would require a huge amount of photons in order to approximate the illumination coming from specularly relevant directions.

For caustics, both an accurate and approximate calculation method also exist. The accurate calculation uses the caustics photon map as a radiance estimate. The approximate evaluation just assumes that the contribution from caustics is included in the global photon map.

Finally, a "multiple diffuse reflections" term is calculated. The approximate calculation of this term comes from the global photon map. The accurate evaluation is done by performing Monte Carlo ray tracing. Even if a more accurate calculation is chosen, the values in the photon map can be used to give get an estimation of which directions are most important to sample when generating the Monte Carlo samples. The directions associated with the photons contained in the photon map represent directions where indirect illumination is likely to come from.

Photon Mapping, while a great estimation of the interaction of light within a scene, is ultimately just an extension to ray tracing and is not particularly useful in real-time applications. If enough photons were to traced through the scene, the global photon map might be an accurate enough representation of the illumination in the scene, but sampling this in real-time would be a painful experience with that many photons. Regardless, the results that can be achieved using photon mapping are often pleasant looking, making it a useful extension to offline renderers.

3.2.2 Real-Time Photon Mapping

Since the creation of photon mapping, several attempts to make it a real-time technique have emerged. One such way of doing so was outlined in the paper *Real Time Photon Mapping* [Jozwowski02]. The main modification to the algorithm in this case is that rather than shooting tens of thousands of photons into our scene, only perhaps one thousand total photons are shot out. Additionally, rather than shooting a whole new set of photons every frame, only a fraction of the total amount of photons are emitted every frame and a majority of the photons that get used are from several frames ago. Since many of these photons are old, they may not necessarily correspond to the current state of lighting, but there is usually enough congruency between frames that this slight delay in updating the photon map shouldn't produce terribly noticeable artifacts. As new photons are shot, old ones are deleted, maintaining the number of photons in the map at a constant number. Finally, a radiance estimate can be made at each vertex in the scene, and the scene can be rendered by a conventional graphics API like OpenGL.

This algorithm can be summed up as "do less so that it takes less time". A traditional KD-tree is used, like in the original algorithm, but some extra work must be done remove photons based on when they were created. An additional array is used to store photons based on when they were created and map to their location in the KD-tree in order to quickly find and remove photons in the KD-Tree. Also, the maximum amount of diffuse interreflections is fixed so that the maximum amount of intersections that need

to be calculated is known ahead of time. The most expensive part of this algorithm is the photon intersection tests. Typical intersection optimizations are used, such as using bounding volume hierarchies and using simple imposter geometry to make intersection less computationally expensive.

The visual results presented in the paper are only moderately pleasing. Indirect illumination can gets produced, but it looks very unnatural. The algorithm can run in real time, but the complexity of the scenes that can be rendered as well as the lighting resolution (number of photons) that can be used is very limited. However, the technique was developed and tested on old hardware (about 8 years old at this point) so there is a chance that more modern hardware could help enhance the results of this technique. Regardless, it is probably not very useful in a real-time game application.

3.2.3 Image Space Photon Mapping

More recently, a technique known as "Image Space Photon Mapping" has been developed. [McGuireLuebke09] This technique expands upon photon mapping by using modern rasterization hardware to expedite certain aspects of the photon mapping pipeline. Three aspects of traditional photon mapping are recognized as possible avenues for optimization:

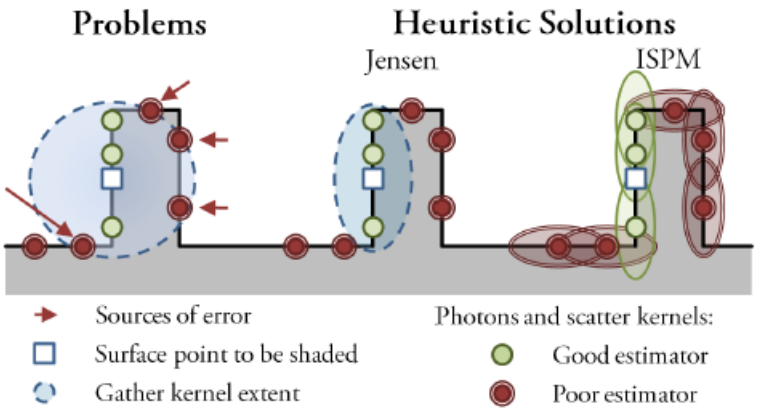
1. *The initial bounce of Photon Mapping is expensive because it typically requires handling the greatest amount of photons.*
2. *The final bounce of Photon Mapping is expensive because in traditional Photon Mapping the photons are gathered at every pixel, requiring a search over all existing photons.*
3. *The first and last bounces of photons have the same center of projection, which means that it is possible to take advantage of rasterization to compute them.*

With these observations in mind, the authors introduce two concepts: The Bounce Map and the Photon Volume. The algorithm works for point light sources, but not for area light sources. It has four distinct stages

1. *Rasterize G-buffers from the eye's point of view for deferred rendering.*
2. *Rasterize a bounce map from each light's point of view.*
3. *Trace bounced photons on the CPU using world-space ray tracing.*
4. *Scatter the final photons in screen space using the G-buffers and rendering photon volumes.*

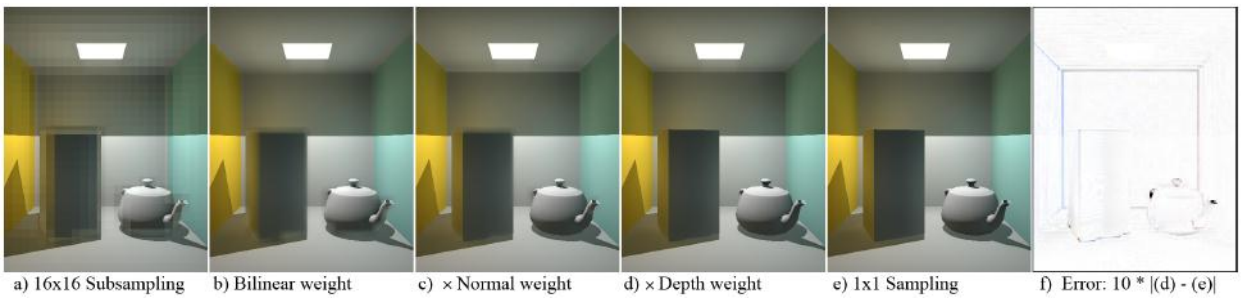
The first step is self-explanatory. The idea of the bounce map in the second step is more interesting. At a high level, it involves treating each rendered pixel in the light-space g-buffer as a photon (This is very reminiscent of Reflective Shadow Maps and Splatting Indirect Illumination). Each photon stores a position, power value, and direction that represent the state of a photon directly after being bounced off of a surface. In this way, the first bounce is calculated on the GPU. If a photon is absorbed, a power value of zero is output, and its direction is undefined. This algorithm counts on the fact that a large amount of photons will be absorbed in the first bounce phase, meaning that when the subsequent CPU bounces are calculated there will be fewer photons to deal with.

The final step involves rendering a photon volume (a skewed icosahedron in their implementation) at the location of all photons that survive the first bounce. As these photon volumes get rasterized, the lighting contribution of the photon being represented by that photon volume is scattered to all of the pixels within that conservative photon volume using a Gaussian kernel. This shows an interesting departure of this algorithm from traditional photon mapping in that traditional Photon Mapping gathers photons at each pixel whereas ISPM scatters photons in image space. This change of paradigm from gathering to scattering eliminates the need for the expensive kd-tree balancing and searching.



An example of fitting the Photon Volumes around the photons. On the left we see that a sphere is not a good estimator because it allows for the possibility of covering a lot of unnecessary surface points. The ISPM heuristic of skewing the photon volume along the surface it is on provides a better estimator. (Image taken from [McGuireLuebke09])

One final point about the rendering process is that trying to sample the radiance contributions of the photons at full resolution can be quite costly for high resolution images. As such, it can be more beneficial to apply the screen-space "splat" of the photon volumes onto a lower resolution buffer and upsample it using various geometry-aware filtering methods. In other words, performing filtering that takes into account the normal and depth discontinuities in the samples being used to upsample. This can provide a large speed increase at the cost of producing improper results.



An example of different subsampling/smoothing methods. a) shows the result of 16x16 subsampling, where every sample represents 16 pixels in the full resolution image. Images b,c, and d show the result of adding in different weights to the smoothing. Image e) shows a reference rendering where no subsampling is performed. (Image taken from [McGuireLuebke09])

The results are compelling. The performance numbers presented in the paper indicate that it is possible to render 161,612 polygons at 1920x1080 resolution, and 4x4 subsampling at 26.5 fps, which is impressive considering the quality of the results. It does, however, suffer from the same problem as shadow volumes where, if the camera

is inside of a photon volume, the contributions of that photon will not be properly rendered. Aside from that, this algorithm does a good job of providing single to multiple-bounce indirect illumination in real time.

3.3 Instant Radiosity Solutions

Instant Radiosity is a method that is similar to Photon Mapping with a few key differences. In a first pass a series of virtual point lights are generated. These Virtual Point Lights (VPLs) can be generated in a variety of different ways, with Photon Mapping-based ray tracing being one of them. Other real-time methods generate VPLs via rasterization and the traditional graphics pipeline. Once a set of VPLs has been generated, they can be used to light the scene. To this end, each VPL is treated as a new light source.

3.3.1 Classic Instant Radiosity

The first paper to describe Instant Radiosity was written by Alexander Keller [Keller97]. The basic premise behind Keller's algorithm is that several VPLs are generated by tracing photons out from each light source out into the scene. Direct lighting is then computed for the scene, with full lighting from each VPL, including shadows, also being computed. Calculating realistic looking shadows and lighting for each VPL is not a real-time operation for the amount of VPLs necessary to provide a good looking solution, even if modern techniques like shadow mapping are used. Additionally, emitting photons from the light to create the locations of the VPLs require a ray-tracing solution to perform intersection tests with the environment. As such, classic Instant Radiosity is not a real-time algorithm. Other techniques, however, have used these fundamental ideas to develop IR-based techniques that do perform in real-time.

3.3.2 Reflective Shadow Maps

In an attempt to make a real-time IR solution, Dachsbacher and Stamminger developed the technique of Reflective Shadow Maps (RSMs) [DachsbacherStamminger05]. RSMs attempt to model the first bounce of indirect illumination in a scene using the same principles behind Shadow Mapping. It is an image-space technique in that its algorithmic complexity is not dependent upon the geometric complexity of the scene being rendered. It is only an approximation of indirect illumination as occlusion is not considered for the indirect illumination, which can result in rendering artifacts. However, the results are in many cases plausible, which is often sufficient for indirect

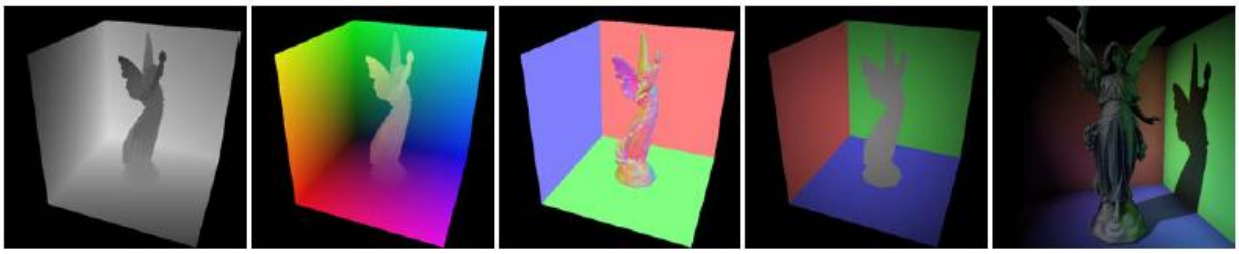
illumination. RSMs build upon the authors' previous work of Translucent Shadow Maps [DachsbacherStamminger03], where the pixels of a shadow map are also considered to be new point lights in the scene.

3.3.2.1 RSM Data

A Reflective Shadow Map works like a normal shadow map does, but the difference lies in what data gets generated. The scene is rendered from the point of view of the light, and instead of just rendering a depth map, the authors also store the world space position, the surface normal, and the reflected radiant flux of each visible surface point. This is done in a single pass via Multiple Render Targets (MRTs). While the world space position could be reconstructed from the depth value, they prefer to have the world space position available to save shader instructions later on (Shader ALU work is one of the bottlenecks of the process). With this data, they then have all of the information necessary to perform a lighting calculation, treating each pixel in the RSM as a point light using the following equation:

$$E_p(x,n) = \Phi_p \frac{\max\{0, \langle n_p | x - x_p \rangle\} \max\{0, \langle n | x_p - x \rangle\}}{\|x - x_p\|^4}$$

This formula is defined as "The irradiance at a surface point x with normal n due to pixel light p."



The different buffers that make up a Reflective Shadow Map. From left to right: depth, world space coordinates, world space normal, flux, final image with indirect illumination applied. The first four buffers are generated in a single pass using Multiple Render Targets and rendered from the light's point of view. (Images taken from [DachsbacherStamminger05])

3.3.2.2 Implementation

In order to generate all of this data, the authors use multiple render targets in a single pass when rendering the RSM. The position and normal data can be calculated in any

coordinate space, but it makes sense to store everything in world space, as that is where all of the calculations will likely be done. Flux is calculated as the amount of light going through a pixel times the reflection coefficient of the surface at that pixel. It is noted that problems tend to exist at the common boundary of two walls. This is because the illumination integral approaches a singularity, which is hard to handle numerically. To fix this issue, the authors suggest moving the pixel lights in the negative direction of the normal at the pixel. While this may cause problems for very thin objects, it is a possible enhancement due to the fact that occlusion is not considered for this algorithm.

Once all of the RSM data is rendered into textures, the indirect illumination can be gathered from the pixel lights. The scene is rendered again from the point of view of the camera, and for every visible pixel the contribution of all of the virtual point lights in the RSM is determined for that pixel. Ideally every pixel in the RSM would be considered to be a VPL, but this is incredibly computationally expensive, prohibitively so for a real-time application. Consider that a typical shadow map is fairly large, (the paper mentions 512x512, but this is perhaps small by today's standards) and that attempting to read all of the pixels in a shadow map at every pixel in our camera view would potentially require $(width_{rsm} * height_{rsm} * width_{image} * height_{image})$ lighting calculations to be performed. In order to preserve the real-time nature of the algorithm, a subset of the RSM pixels can be sampled. The closer a sampling pixel is to original pixel, the more relevant it will be. To that effect, texels can be sampled in a disc around the original point, weighing closer pixels as more important. To produce a uniform sample in a circle, two uniform variables are used to produce polar coordinates relative to the original pixel. The paper suggests that it is likely sufficient to use 400 VPLs at each pixel in the indirect illumination buffer.

In order to produce a sample point, the following formula is used.

$$(u, v) = (s + r_{max}\xi_1 \sin(2\pi\xi_2), t + r_{max}\xi_1 \cos(2\pi\xi_2))$$

Where (s, t) are the original sample points, r_{max} is the maximum sampling radius (in the range $[0,1]$ and $\xi_{1,2}$ are two uniform random variables. To compute these points, the offsets $(r_{max}\xi_1 \sin(2\pi\xi_2),\ r_{max}\xi_1 \cos(2\pi\xi_2))$ can be calculated once at the beginning of the application and reused for the entire program execution. Once the 400 or so sets of coordinate offsets are produced, they are stored in a texture and used as a lookup table in the pixel shader during the VPL gather phase to sample into the RSM.

Once the sample offsets are calculated and the RSM has been created, the gathering phase can be performed. The scene is rendered from the point of view of the camera, and at each pixel the texel in the RSM that corresponds to our current pixel is determined (much like normal shadow mapping). That texel's coordinates are used in conjunction with the precomputed offsets to look up into the RSM the 400 or VPLs around the current texel's position. For each lookup, the above equation is used to determine how much light is contributed from each of the 400 VPLs at the current point. The contributions of each VPL are summed up and stored in an intermediate indirect illumination texture to be used later when lighting the scene. This is unfortunately still a very expensive operation to do every frame. To try and speed things up, the VPL gather phase can be performed at a lower resolution than the final scene resolution. Then, when combining the indirect light with the final scene, a screen-space bi-linear interpolation method is used to upscale the low-resolution indirect illumination buffer to a normal-res buffer. This bi-linear interpolation can only be performed on pixels that have similar normals and/or depths. Otherwise, unwanted artifacts can occur where indirect illumination from one object can fall onto objects that should not receive it. For any pixels for which the interpolation scheme will not work, the indirect illumination must be performed again, this time at full resolution.

The authors used the RSMs in a deferred shading environment, where the indirect illumination gather phase is performed in a single screen-space pass, but it is possible to perform it in a forward-rendering environment. To do this, a method to map each pixel that is being shaded into the light's view space is necessary in order to be able to find

the original texel's location in the RSM. Similarly, one can apply the results of the indirect illumination regardless of what lighting model is used. One thing to note is that RSM is a technique that is performed for a single light at a time. It seems common to have one global directional light that produces indirect illumination, and several local point lights that only produce direct illumination. If it is desirable to calculate indirect illumination for every light in our scene, it is necessary to render an RSM for every light (if the light is a point light it would require a cube RSM), and perform the VPL gather phase for every light, which is very expensive both computationally and in terms of memory usage. However, the technique can reasonably be applied for directional lights and spot lights.

RSMs can be classified as an Instant Radiosity method because it has two phases: A VPL generation phase, and an indirect illumination calculation phase that uses the VPLs. In this case, the VPL generation phase is the RSM creation step combined with the sample offset creation phase. Every pixel in the RSM is a VPL, regardless of if whether it is used or not. The calculation phase uses the VPLs (RSM texels) to calculate indirect lighting at every pixel (or perhaps at a lower resolution than the framebuffer.)

The timing results of RSM are good enough to be considered real time. For various combinations of RSM resolution, number of VPL samples, and indirect illumination buffer resolution, the process can achieve anywhere from 15 to 30 fps in their testbed. Since graphics hardware is always improving, these results will likely improve in the future. Being a screen-space method, it does not have to be tied to the geometric complexity of the scene being rendered. The process has two different bottlenecks: ALU computation and memory bandwidth. First of all, doing hundreds of VPL gathers at each pixel clearly requires a lot of work. This can only be improved by using fewer samples, but doing so causes the quality of the indirect lighting to degrade. Improved hardware can help alleviate this bottleneck. Regarding memory bandwidth, there is the necessity to sample from the three RSM textures for every VPL calculation. If this done in conjunction with a deferred shading approach, it can add as many as four additional

texture samples (seven total) that need to be performed in order to calculate a single VPL's contribution. Even with all textures resident in video memory, this is a lot of data that needs to be pulled into every shading unit at every pixel, so it is clear that memory bandwidth can quickly become a problem if one is not careful. By tweaking the various parameters of the algorithm, it is possible to achieve good looking results in real-time.

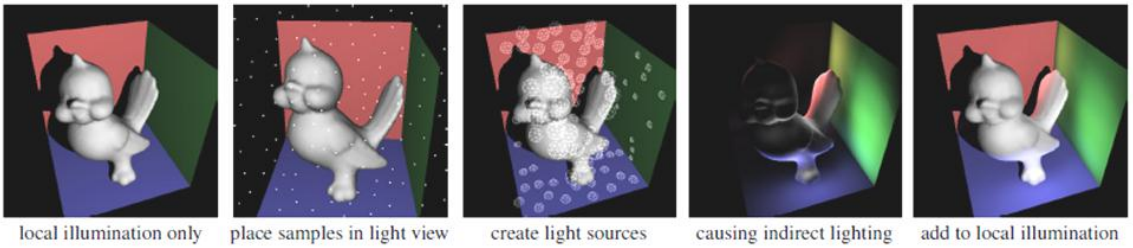
3.3.3 Splatting Indirect Illumination

Splatting Indirect Illumination is a technique that was developed by the same researchers who developed Reflective Shadow Maps [DachsbacherStamminger06]. It is based on RSM, and presents many of the same advantages and disadvantages that were present in their earlier research. Like RSM, it allows for dynamic light sources and dynamic geometry, and can be extended to the rendering of caustics and glossy reflections. Only the first bounce of light is calculated, and occlusion for the indirect lighting is not taken into account.

3.3.3.1 Implementation

Splatting Indirect Illumination (SII) starts by generating a traditional RSM, without any deviation from the original algorithm. A flux, depth, normal, and position map from the point of view of the light are required. Then, a deferred shading approach to apply local (direct) illumination to the scene is performed before starting to calculate the indirect illumination. Next, a constant set of screen space coordinates stored in a texture. These coordinates will represent the locations of VPLs, and can be generated uniformly, or by using an importance sampling approach. At each coordinate, a screen-space quad is rendered to cover all of the pixels that might be influenced by that VPL. At each camera-space pixel covered by the screen splat, the original sampling point's data from the RSM is used to compute indirect lighting. The results are rendered into an

accumulation buffer, which is later used to calculate the final scene lighting.

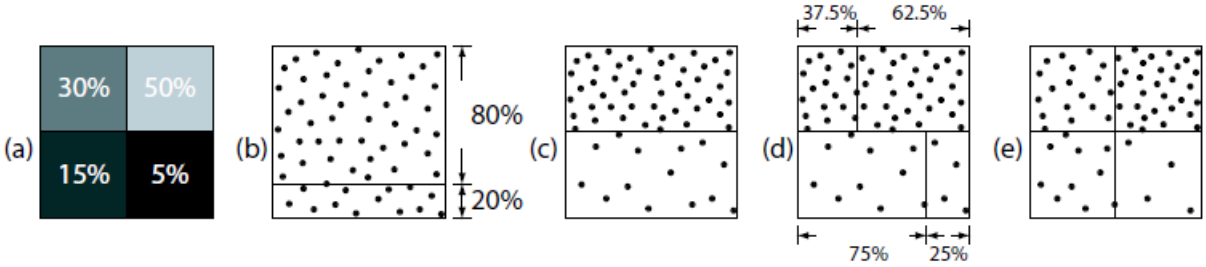


An example of the different phases of SII. First, local illumination is calculated. Next, sample points are chosen from the RSM and projected into world space. These light sources are evaluated and the indirect lighting they produce is calculated. Finally, this illumination is added to the local illumination. (Images taken from [DachsbacherStamminger06])

Being so similar to Deferred Shading in its lighting calculation phase, some of the same optimizations that are possible in Deferred Shading can be applied to the VPL calculation phase. For instance, rather than rendering a screen quad, it is possible to render a convex shape to represent the hemispherical lobe of indirect illumination originating at a surface in order to reduce the amount of overdraw that we incur. Since the algorithm is likely bound by its pixel operations, this is a useful optimization. Additionally, drawing geometry in world-space rather than screen space allows for the utilization of Z-testing to potentially save calculations on pixels that are hidden from view.

In order to place the VPLs in more strategic locations than would result from simple uniform sampling, the paper uses a form of Importance Sampling by hierarchical warping. As a probability distribution, the normalized flux of the scene is used. The process starts with the original set of uniformly distributed screen space samples $(x, y)\epsilon[0,1)$ with sample weight 1. The set of samples are moved first in a vertical step, then a horizontal step. If the lower half of the screen samples account for 20% of the flux, the points will be scaled such that 20% of the samples are located in the lower half. Similarly, if 25% of the samples are located in the right half, then the samples are scaled so that 25% of the samples are located in the right half. In this way, more samples are allocated where the most flux is, rather than landing in areas where no lighting is occurring. The sample weight of each sample is modified so that the total weight of all samples remains equal. When the time comes to extract data from the RSM, the

sample weight is multiplied against the flux in the RSM to determine the strength of that sample's indirect illumination.



An example of the sample warping via importance sampling. This helps ensure that more samples are allocated where more radiance is likely to come from. (Images from [DachsbacherStamminger06])

The authors spend a brief moment considering the impact of ambient occlusion upon their algorithm. They propose to scale the flux in the RSM by whatever ambient occlusion term was calculated for that sample's point. The logic behind this is that the amount of indirect illumination that would be leaving a surface is proportional to the amount of incident light at the surface, a quantity which ambient occlusion happens to approximate reasonably well. If two samples receive the same direct illumination, but have differing levels of ambient illumination, they will produce unequal levels of indirect illumination.

The results of SII are comparable to those of RSM in terms of quality of the indirect illumination generated. It is generally a less computationally expensive algorithm because it greatly reduces the amount of texture fetches/memory bandwidth that occur at every pixel. It produces indirect illumination for dynamic scenes, with dynamic lights, and dynamic cameras, with the only drawback being that the indirect light is not occluded by other geometry.

3.3.4 Multiresolution Splatting for Indirect Illumination

Multiresolution Splatting for Indirect Illumination [NicholsWyman09] seeks to improve upon the algorithm of Splatting Indirect Illumination. SII was designed as a scatter analogue to the gather-based RSM technique. However, fill rate still tends to be a bottleneck in SII. Each VPL in the scene must be splatted in screen space, with each VPL covering potentially a large amount of pixels. Rather than simply reducing the size of

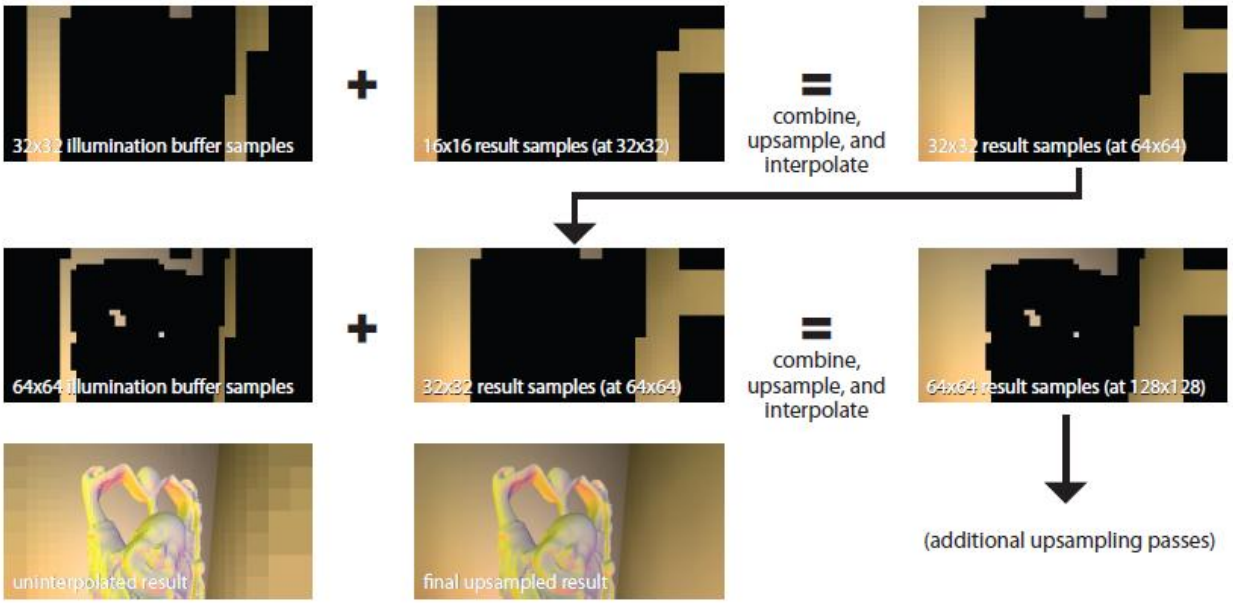
the indirect illumination buffer to reduce the amount of pixels that require shading, this paper presents the idea of performing splatting on several multi-resolution buffers, depending on the behavior of the underlying geometry. It is inspired by the work of Wyman with his various hierarchical techniques, including Hierarchical Caustic Maps [Wyman08]. It makes use of a min-max mipmap to help gather hierarchical information about a scene at many resolutions.

The first step in this algorithm is to render a classic RSM from the light's point of view, along with a set of G-buffers from the camera's point of view for deferred shading. The RSM is sampled on a regular grid to select the VPLs that will be splatted into the screen. Next, a min-max mipmap chain is created for the scene. Only mipmaps for the Normal and Depth buffers need to be generated, as these are the most useful in discovering discontinuities in the scene across which indirect illumination cannot be computed at a low resolution. For the Normal mipmap, a separate min-max mipmap for each coordinate of the normal is generated, and a discontinuity is considered to occur whenever ANY of the three coordinates differs greatly.

Splatting starts at the lowest resolution mipmap, which is 16x16 texels in size. At this point, a single full screen splat is used, as the dimensions of this mipmap level are so small. Each subsplat is represented by a point primitive, and can either get rendered into the current level of the mipmap, or refined into four subsplats at the next lowest resolution. Refinement occurs whenever the difference in the min/max value stored in the min-max mipmap exceeds a certain threshold. This continues until no more discontinuies are discovered, or the highest resolution of splat size is achieved. Each VPL can have its own subsplat refinement pattern, or the refinement can be calculated once per frame and shared across every VPL.

Each subsplat is stored as a three-tuple: one value indicating the output resolution of the subsplat, and two values for the screen-space location of the splat. At render time, a vertex shader positions each subsplat into the correct layer of the illumination buffer(which is just mipmapped normally), indirect illumination is accumulated, and

additively blended into the appropriate illumination buffer mip level. Once all of the splats are rendered, upsampling must occur. Upsampling is performed one mip level at a time. Starting at the second coarsest mip layer, the next-highest resolution mip level is sampled and its contributions blended into the current layer. This happens all the way down the mip-chain until the highest resolution buffer is reached. Empty texels are not considered when interpolating values, i.e. only texels that already had data in them at the start of the upsampling will be considered for interpolation.



An example of how the upsampling process occurs. Lower resolution images are upscaled and combined with higher resolution images until the entire mip chain has been reconstructed. (Image taken from [NicholsWyman09])

3.3.5 Imperfect Shadow Maps

Imperfect shadow maps (ISM) are a way to calculate real-time indirect illumination of large and fully dynamic scenes. [RGKSDK08] By combining the idea of Imperfect Shadow Maps with the principles of Instant Radiosity, it is possible to create convincing looking occluded indirect illumination. As previously mentioned, one of the most expensive parts of attempting to model indirect illumination lies in the need to calculate visibility and occlusion for multiple bounces of light. Visibility for direct lighting can be determined using methods like shadow volumes and shadow maps. However, once the direct lighting hits a surface, light gets reflected in every direction (from diffuse

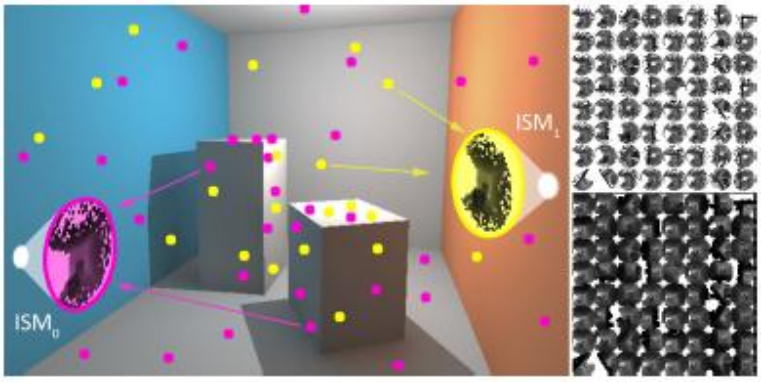
surfaces) and it is nearly impossible to sample continuous hemispheres of light at many points in a scene at real time. Imperfect Shadow Maps makes the observation that accurate calculation of visibility is not usually necessary to create convincing indirect lighting, and imperfect information can still lead to decent results. The algorithm is independent of scene complexity, and requires minimal precomputation in order to work

3.3.5.1 Preprocessing

The only precomputation involved in ISM involves approximating the scene to be represented by a series of points with roughly uniform density. Triangles are picked out of the scene with probability proportional to the area of the triangle, and a random point inside the triangle is chosen. More sophisticated methods of picking triangles end up not being necessary. The points are represented by their barycentric coordinates relative to the triangle and the index of the triangle in the scene.

3.3.5.2 Creating ISMs

The creation of an ISM involves rendering the points created in the preprocessing step into a very low resolution shadow map. The points used to represent the scene are rendered as hardware point primitives (e.g. GLPoints), and the size of the point is scaled based on the squared distance of that point to the corresponding VPL position. Since the ISM needs to hold a whole hemisphere of information, parabolic maps are used. For dynamic objects, the points are transformed to their current locations before being used. A large amount of these low-resolution ISMs are created and rendered in one pass into a single large texture. The set of points is split up amongst all of the ISMs in a vertex shader and rendered into a large texture. Each ISM receives a fixed, random subset of the point set. As an example, ISMs can be rendered at a resolution of 128x128 pixels each and stored in a 4096x4096 texture.



*An example of Imperfect shadow maps. On the left, two ISMs are rendered, with the pink points representing geometry that gets rendered to ISM_0 and the yellow points being rendered to ISM_1. On the right, we see two example of 64 ISMs being stored in a large texture. The top set does not use the push/pull method of reconstruction, and the bottom one does. (Images taken from [*RGKSDK08])

The low resolution of these maps combined with the sparsely sampled environment can create holes in the environment where geometry should be. To fix this, a push/pull phase is applied. [GrossmanDally98] In the pull phase, an image pyramid is created where the image is downsampled by a factor of two. To determine which pixels should be used to contribute to the coarser levels, only samples that are close to each other are combined (outlier rejection). In the push phase, pixels from the coarser levels are used to fill in holes in the finer levels. A pixel in the finer level has its depth value replaced if the value obtained from the coarser level greatly differs from the existing pixel value. Separate depth thresholds are used for the pull and push phase, which get scaled by an exponential factor according to the current mip level. Typically the threshold values are set to 5% of the scene extent. In most cases, going two levels up in the mip chain is sufficient to fill in all of the holes in an image.

A single ISM gets rendered for each VPL. Any method can be used to generate the VPLs in the scene, but the original paper proposes a way to generate them on the GPU. The 3D position of each VPL is determined by rendering a cube map from the viewpoint of the point light source, which is importance sampled, and n VPLs are chosen from among the pixels in the cubemap, very similar to how SII chooses to place its VPLs. For each VPL, a parabolic ISM is rendered, oriented along the normal of the surface. These ISMs are then used to light the scene with each of the VPLs, producing occluded indirect illumination in real time.

ISMs can be extended to multiple bounces by treating them like RSMs. The first set of VPLs are generated, and many ISMs are rendered. Each ISM is importance sampled, and a new set of VPLs is generated, producing another set of ISMs. The process can continue for however many bounces of light are necessary.

Results of testing by the authors show that 128x128 is an optimal resolution for ISMs in the case where parameters will not be adjusted by hand. Going up to 256x256 does not significantly lower the error compared to a path-traced reference image. Temporal flickering can occur when too few VPLs are used (which truthfully is a shortcoming of the original Instant Radiosity method).

3.3.5.3 Interleaved Sampling

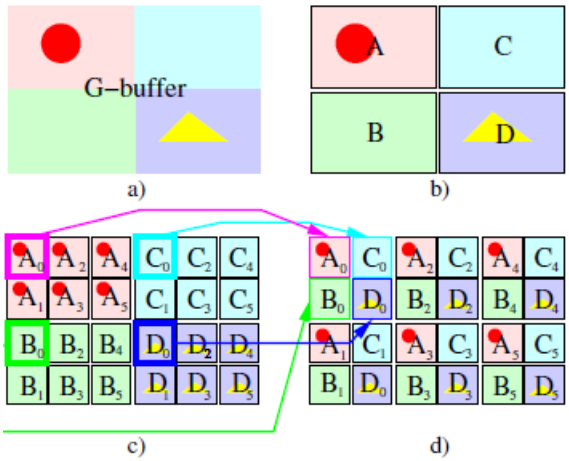
The original implementation of Imperfect Shadow Maps integrates a useful technique to help optimize the scattering of light from the VPLs. Rather than needing to determine the contributions of every VPL at every pixel, the scene is split into several smaller GBuffers, and a subset of the VPLs are used to light each subbuffer. Once all of the subbuffers have been illuminated, they are recombined into a single buffer, and the results are blurred using a geometry-aware blur. This technique is known as Interleaved Sampling [SIMP06].

To more clearly describe this technique, imagine a standard deferred shading implementation. Typically, lighting calculations will be performed at every pixel within the range of a light's influence. Interleaved Sampling proposes creating an NxM pattern (e.g. 2x3) such that every pixel in a NxM block (6 distinct pixels in our case) is only lit by a mutually exlusive subset of all L-many lights in the scene. For instance, in a 2x3 pattern, each pixel will be lit by $L/6$ lights, and the contributions of each light will get spread to neighboring pixels via a geometry-aware blur.

A naïve implementation of this algorithm might set up a stencil buffer such that one pixel in each NxM tile gets activated for shading at a time. This will succeed, but it is not at all texture-cache-friendly considering all of the G-buffer accesses that need to be performed. To remedy this situation, the authors propose rearranging the G-buffers

such that all of the pixels in each NxM pattern that need to get lit by a certain light set are placed next to each other and can be accessed in a linear fashion, rather than requiring the GPU's texture unit to jump all around the g-buffers to find the relevant pixels. This splitting process results in a single G-buffer being split into NxM tiles that look like the original image, but each tile only contains the pixels relevant to a certain light set, and all of the pixels in each tile is unique.

Splitting the G-buffers can be achieved in one pass, but a one pass algorithm suffers from the same cache-unfriendliness that the original naïve implementation suffers from, so the authors choose to instead make the splitting process more cache-friendly by performing a two pass method. Once the G-buffers are split, each set of lights is applied to the appropriate one of the NxM tiles that it was assigned to. After all of the shading calculations are done on the tiles, the original texel order is restored for the lighting buffer, producing an interleaved sample pattern. This reconstruction pass is also a two pass algorithm that is just the reverse of the two-pass splitting algorithm.



An example of splitting a buffer into a 3x2 sample pattern using the two-pass algorithm. a) shows the original G-buffer configuration. In c) we split up each of the four subregions in b) into a 3x2 subpattern. We can then translate these blocks into our intended 3x2 sample pattern in d). This intermediate translation is intended to make use of cache coherency when translating pixels to/from our sample pattern. (Images taken from [SIMP06])

The paper describes the steps involved in a deferred shading process using Interleaved Sampling:

1) G-buffer Creation: This is the same as in normal deferred rendering. Positions, normals, and color are output to separate render targets of the same resolution as the target image.
2) Buffer Splitting: In this step, the G-buffers are split into the NxM tiles.
3) Shading Computations: Lighting is performed on each NxM block. Normal deferred shading can be performed by just setting the viewport to cover only one of the $N * M$ blocks at a time.
4) Buffer Gathering: The lit pixels are rearranged to resemble the original image using the opposite of the two-pass algorithm in step 2. In this new lighting buffer, each pixel in an NxM block contains lighting data for a unique subset of lights.
5) Filtering: Discontinuity Buffers: This pass creates a discontinuity buffer to determine where the edges of objects are.
6) Filtering: Gaussian Blurring: This pass uses the discontinuity buffer from step 5 to perform a geometry-aware Gaussian blur across the lighting buffer. As the separable Gaussian Filter is being applied to the image, samples are only used until a discontinuity is reached or the kernel size is reached. This step can take advantage of hardware filtering of textures to reduce the number of samples that need to be taken in order to blend the contributions of nearby texels.
7) Lighting Application: The contents of the lighting buffer are then modulated with the surface albedo colors to produce the final, lit image.

This algorithm produces very good results. The authors provide performance numbers that indicate that they experienced an order of magnitude speedup by using this method of sample interleaving. For 480 point lights, they were able to achieve frame rates of about 36f/s with interleaved sampling, and only 1.2 f/s without interleaved sampling.

Interleaved Sampling can be applied to almost any Instant Radiosity-based method in order to speed up rendering. The only requirement is that you have some set of lights

(VPLs, Direct Lights, etc.) that you can distribute amongst an interleaved pattern. This is a useful optimization because for each light, one only has to invoke the pixel shader at some subset of the original image's pixels, resulting in less computation time required.

3.3.6 Clustered Visibility

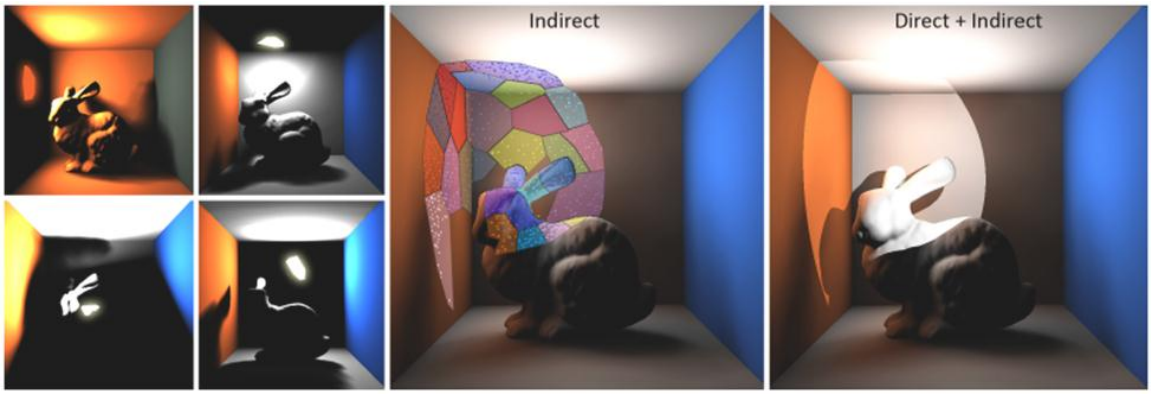
Clustered Visibility is another Instant Radiosity-based attempt to solve the problem of visibility for indirect illumination [DGRKS09]. The goal of this technique is to render soft shadows for large amounts of VPLs by grouping VPLs into clusters and determining visibility for an entire cluster rather than for individual VPLs. In this way, VPLs are treated as VALs (Virtual Area Lights), and soft shadows can be computed for indirect illumination. All VPLs contribute to the illumination of the scene, but all VPLs in a VAL share the same visibility information. The algorithm allows for the use of any soft shadowing algorithm to calculate visibility for the VALs. The authors chose to use Convolution Soft Shadow Maps to compute percentage-closer filtering that adapts to the shape of the light.

At the center of each cluster, a VAL is placed and a CSSM is rendered. It is only necessary to gather a single hemisphere's worth of CSSM data. As such, a parabolic projection is used for the CSSM. In order to perform the clustering, something known as k-means clustering is used. They choose to cluster VPLs that have similar normal and positions based on the surface point that represents each VPL. K-means clustering begins by using arbitrary cluster centers and assigning each VPL to the cluster to whose center it has the smallest distance. Next, cluster centers are recomputed as the average of all positions in the cluster. This is repeated to convergence. Normal k-means clustering only takes distance between points into account, so in order to incorporate surface normal as a criteria it is necessary to define distance as a weighted sum of Euclidian distance and angle between VPL normal and cluster normal. Normals are taken into account to prevent clusters from crossing planar boundaries, which would result in incorrect visibility information later on. Clusters are recomputed every frame.

The paper uses RSMs as the method of generating VPLs, upon which a Halton sequence is used to sample for N VPLs. A subset of M VPLs among the N VPLs is chosen to be the center locations for each of the M clusters. Clustering is then performed on the GPU. Each VPL is represented by a vertex, which gets rendered into 4 M-Texel lookup textures representing position, normal, irradiance, and count. For each VPL, the distance (including normal) is calculated between the VPL and each of the M clusters. The vertex is then scattered to a location such that the point primitive will be rendered to the appropriate texel in the M-texel textures. Colors, positions, and normal are additively blended, and a final pass divides each texel in the M-texel textures by the value of the texel in the count texture associated with that VAL. As such, an average of all VPLs is computed on the GPU. Each VPL is then mapped to a VAL by again looping over the M-texel position buffer and determining the VAL to which the VPL has the smallest distance.

Shadow maps are then computed for each VAL and stored in a texture array. When it comes time to compute indirect illumination, interleaved sampling is used to lessen the per-pixel load, and the appropriate VAL shadow map is used to determine visibility for each VPL.

The algorithm suffers from a few problems. Particularly, using Reflective Shadow Maps means that the algorithm is restricted to point, spot, and directional lights. Also, since VALs use rather low resolution shadow maps, it is possible to miss shadowing from thin occluders. The results look very good and are calculated in real-time.



Results from Clustered Visibility. The four images on the left show individual shadow maps from some of the VALs that are created. The middle image shows the clustering of the VPLs within each VAL (M = 30). Each of the VALs has a shadow map computed for it, and each VPL uses the shadow map of the VAL that contains it. The image on the right shows the results of lighting with each VPL. (Images taken from [DGRKS09])

3.4 Other Solutions

The following techniques are varied in nature, and it is difficult to choose which category they belong in. As such, they will be presented together in a separate group.

3.4.1 Precomputed Radiance Transfer

Precomputed Radiance Transfer [SKS02] is a concept introduced as a way to render global illumination in real time with some precomputation. It uses 3^{rd} to 5^{th} order spherical harmonics to represent lighting, either local or global. For an explanation of spherical harmonics, please see Appendix A. An overview of the algorithm is as follows

1) In a preprocess step, project a transfer function onto spherical harmonics for various points over the surface of a model. For purely diffuse reflection this results in a set of *transfer vectors*, and for glossy surfaces this becomes a set of *transfer matrices.*
2) At run-time, project incident radiance onto SH basis functions.
3) Use these transfer vectors/matrices and the incident radiance SH functions to compute the final shading.

One piece of notation: a *transfer function* is defined as "an object's shaded response to its environment…mapping incoming to outgoing radiance, which in this case simply performs a cosine-weighted integral." The paper calculates lighting for three different kind of transfer functions: Diffuse Transfer, Shadowed Transfer, and Interreflected Diffuse Transfer.

3.4.1.1 Radiance Self-Transfer

The main idea of this algorithm is to project the incident lighting onto n^2 SH coefficients, where n is the number of bands we wish to use. Lighting is sampled dynamically and sparsely near the surface. This means that for a single model, the incoming light may be calculated for one point across the entire hemisphere, projected onto SH basis functions, and the resulting coefficients will get reused across the entire model. This is a single vector of n^2 coefficients.

Conversely, a set of SH coefficients are precomputed and stored densely across the entire model. This means that the three transfer functions being used (unshadowed, shadowed, interreflected) are precomputed, probably at every vertex of the model, and stored as a large set of vectors (or matrices in the case of glossy surfaces). With incident light represented in terms of SH, and transfer functions represented in terms of SH, it is only necessary to perform a dot product on the two vectors at every point where one wishes to compute lighting on a model.

3.4.1.2 Diffuse Transfer

This type of radiance transfer is for diffuse reflectance at any point on a model, ignoring how a model O reflects or blocks light onto itself. It is an N dot L cosine weighted function of incoming light. Since the object's surface normal are known ahead of time, one can precompute the transfer function by evaluating $N \cdot S$ for many directions S around the positive hemisphere at a certain point p. By doing this, spherical harmonic functions in the direction of S are evaluated and $N \cdot \mathrm{S}$ is projected onto those evaluations. It is noted that second-order Spherical Harmonics are probably sufficient for any arbitrary lighting environment when representing just diffuse transfer, so nine coefficients should be sufficient.

3.4.1.3 Shadowed Transfer

Shadowed Transfer is an extension to Diffuse Transfer in which visibility is taken into consideration. $N \cdot S$ is still calculated many times at each point of a model, but one must now also check if the ray S intersects with O or not. This is called the visibility function $V_p(S) \rightarrow \{0,1\}$, where V equals 1 if the ray S does not intersect with O, and equals 0 when S does intersect with O again. While second order projections were sufficient for unshadowed diffuse transfer, shadowed diffuse transfer is a much higher frequency function, and as such is better represented by either fourth or fifth order spherical harmonics (16 or 25 coefficients).

3.4.1.4 Interreflected Diffuse Transfer

Interreflected Diffuse Transfer is similar to shadowed transfer, but requires the addition of light reflecting from O onto itself. This is where indirect illumination can be calculated for an object. The problem is that this value is not known ahead of time, since the amount of light reflecting off of O at any point is not yet known before calculating either Diffuse Transfer or Shadowed Transfer. In this case, multiple iterative passes are performed, using the results of the previous iteration to accumulate coefficients based on visibility.

There are a few limitations for this algorithm. For interreflected transfer, surface material properties cannot be changed at runtime or inaccuracies will become apparent. One can, however, vary materials for shadowed/diffuse transfer at runtime. Additionally, if blockers or light sources enter into the convex hull of O, inconsistencies will become apparent. The model O can only move rigidly, and cannot deform or animate different pieces relative to itself (does not work for animated models). It is also assumed that there is little variation in lighting across O, since lighting is being sampled sparsely across the surface.

3.4.1.5 Sampling Incoming Lighting Dynamically

A simple way to sample incident radiance at run-time is to sample it at the center of the model O. However, incident radiance can be sampled at multiple points using something called the “Iterated Closest Point” algorithm. If multiple lighting points are to be used, each vertex must now additionally contain a set of weights to blend the contributions from each lighting point. This means one would need to to evaluate lighting for each lighting point and blend the results. However, as everything is being represented as spherical harmonic coefficients, that computation is relatively simple.

In order to dynamically sample the incoming radiance, the paper renders a cube map at each lighting point that excludes the model O during the rendering. The evaluation of the spherical harmonic basis functions are stored in textures related to the cube map, such that each pixel corresponds to the same pixel in the environment cube map.

Rendering these cube maps at a resolution of 4x4 produces only 1% worst case error for 6^{th} order SH, and 8x8 produces .02% worst case error. They tend to use 6 16x16 cube maps for sampling the irradiance. Because it is impossible to do arbitrary inner products on graphics hardware, they have to render out the environment irradiance and spherical harmonic basis function evaluations to textures and perform the projection in software, which tends to be the bottleneck in certain situation.

An example of Precomputed Radiance Transfer. The left image only calculates diffuse lighting, where as the image on the right calculates self-shadowing and diffuse interreflection. (Images taken from [SKS02])

3.4.2 Light Propagation Volumes

Light Propagation Volumes is a technique developed by the company Crytek for use in their game engines. The algorithm makes use of several of the previously mentioned techniques. The main algorithm is as follows:

1. Generate a set of secondary light sources in the scene using reflective shadow maps.
2. Inject secondary light source data into a radiance volume
3. Propagate lighting data throughout the volume
4. Perform final scene lighting with the final light propagation.

These steps will be further described below:

3.4.2.1 Secondary light generation

This algorithm uses Reflective Shadow Maps as a method of generating secondary virtual point lights in the scene. They choose to use RSMs as their method of VPL generation as they claim it is “one of the most highly efficient and highly parallel current

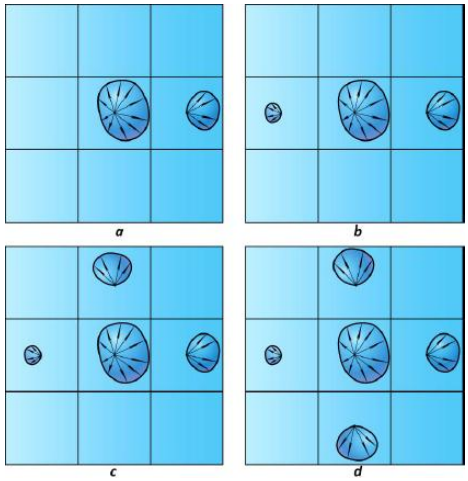
methods for the GPU to sample the secondary light sources of a scene". Since a normal RSM can produce a very large amount of VPLs, they use a smart filter to down-sample the RSM and reduce the amount of VPLs that need to be injected into the light propagation volume.

3.4.2.2 Injection of light data into the Light Propagation Volume

Once the list of secondary lights has been created, these light sources are converted into Spherical Harmonic coefficients, and injected into a three dimensional texture, the Light Propagation Volume. Each VPL that survives the first step is converted into a set of four SH coefficients based on the surface normal of that particular VPL. These coefficients represent a lobe of a second-order spherical harmonic approximation. The vector is scaled by the contribution of the VPL surfel, which is calculated as the product of the light source's intensity, the surface albedo at the VPL, the intensity of the VPL, and the weight of the VPL. The area of the texel represented by the RSM and the volume texture are also taken into account when scaling the contribution. This produces three vectors of SH coefficients, one vector for each band of color (red, green, blue). Each set of coefficients are additively rendered into the appropriate texel of the volume texture based on where it is in the scene.

3.4.2.3 Propagation of light data

The second step calculates an initial light propagation volume and stores it in a volume texture. In the third step, the initial lighting estimate is propagated throughout the entire volume. This is achieved via gathering the contribution of the six neighboring cells at each cell, rather than propagating from a cell to its neighbors. Gathering is preferred over scattering because gathering is a much more logical operation for a GPU to perform than scattering is. At each cell, all six neighboring cells are examined, and the grid-axis-aligned contribution from each neighboring cell is gathered. This has a very interesting consequence that the original directional data of the VPLs is lost, but the full radiance is still preserved in the grid. This process is performed iteratively, as each iteration helps to fully achieve a stable lighting representation of the scene.

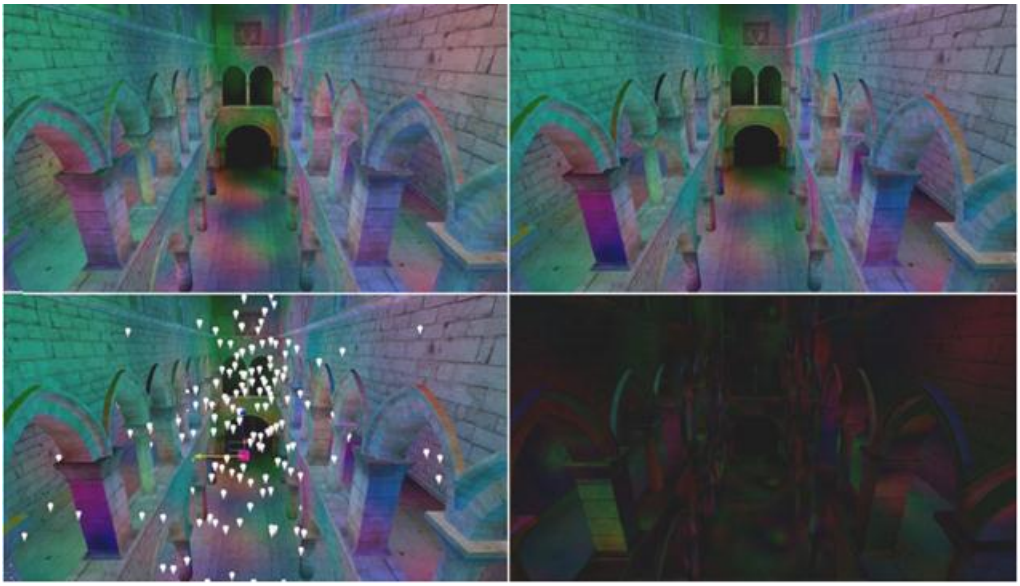


An example of how light is propagated in four iterations from a) to d). Each cell is a grid in the light propagation volume. The symbols in the cell represent a spherical harmonic approximation of radiance. (Images taken from [Kaplanyan09])

3.4.2.4 Lighting the scene

Once the light in the light propagation volume has been propagated a few times, the results can be used to light the scene. This can either be done by looking up the radiance values directly when shading a pixel by projecting that pixel's surface location into the space of the LPV, or if a deferred rendering solution is being used (which CryEngine 3 does), the LPV can be rendered directly into the light accumulation buffer for the final lighting pass.

An interesting usage of the LPV is that it is possible to inject legitimate light sources into the LPV to be propagated out, in addition to just using it for indirect illumination. In this way, it is possible to render many lights in a deferred way without needing a separate pass for indirect and direct lighting. This does introduce a little bit of error compared to the normal deferred approach, but it is usually not too noticeable.



An example of injecting direct lights into a LPV. The white dots in the bottom left image are each individual point lights. By injecting them into the LPV, it is possible to fold massive amounts of direct lighting into the indirect lighting path. (Images from [Kaplanyan09])

Since the LPV represents a finite volume in space, one of two problems can potentially occur. First, it's possible that the LPV grid size is set too small and objects that need to get lit will fall outside of the bounds of the LPV. This would result in them not gaining any illumination that they are supposed to. The other problem that can occur is that if the LPV is stretched to fit the entire view frustum, it is possible that each grid cube will cover too large of an area, and certain objects in each grid might receive inconsistent lighting just because there are other lighting features in that grid. To solve these problems, an extension of Light Propagation volumes has been developed, which is known as Cascaded Light Propagation Volumes. [KaplanyanDachsbacher10] This technique is similar to Cascaded Shadow Maps except it uses several LPVs of varying size. The LPV closest to the camera contains the smallest grid cubes, allowing for the most detail to be present where most of the detail needs to be. As objects get farther from the camera, they eventually fall into larger and larger LPVs. It is fine for these farther LPVs to have larger grids, as accuracy and detail become less important the farther objects get from the camera.

Light Propagation Volumes are a very intriguing way of modeling how light interacts with a scene. It originally did not allow for occlusion of indirect illumination, but as part

of their research into Cascaded Light Propagation Volumes, progress was made into this area with some good preliminary results. Additionally, it can be used to produce glossy reflections if desired, all in real-time. Since it is based on Reflective Shadow Maps, it is mostly independent of the geometric complexity of the scene, and can be used for any combination of dynamic lighting and dynamic geometry.

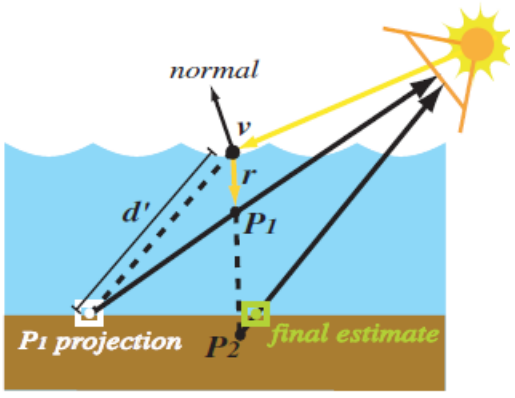
3.4.3 Caustics Mapping

Caustics are an interesting problem in the world of global illumination. Caustics are the phenomena that can be seen when light reflects or refracts off/through an object in such a way that multiple photons/waves hit the same point on a receiver surface, resulting in brighter areas of light in various patterns. This is usually considered a Global Illumination problem because it involves determining what light does after its first interaction with a surface. Calculating caustics in a real-time application is a difficult task because it typically requires the tracing of rays or photons from the light through the refractive geometry, and out into the environment. As always, occlusion and triangle intersection become bottlenecks when attempting to do something like this. The algorithm known as Caustics Mapping is a real-time, image-space algorithm that allows for interactive rendering of caustics that scales independently of scene complexity, and does not suffer from the problems typically faced by ray intersection algorithms. [SKP07]

The overview of the algorithm is as follows: The geometry that is to receive the caustics is rendered to a *positions texture*. This is very much like a typical G-Buffer in deferred rendering. Each pixel in the positions texture represents the position of the geometry underneath that pixel. Next, the refractive geometry is rendered and a position and normal buffer are rendered out. Following that, a grid of vertices is splatted onto the receiver geometry. Each vertex is rendered as a point primitive and constructed such that each pixel from the light's point of view is covered by a single vertex in the grid. In a vertex shader, the vertex is refracted through the geometry and intersected with the receiver geometry using a novel iterative intersection method that runs entirely on the

GPU. The caustic map texture that gets constructed from this is from the light's point of view. At final render time, the caustics map is looked up into by transforming each point (vertex probably) into the light space, and determining texture coordinates from there.

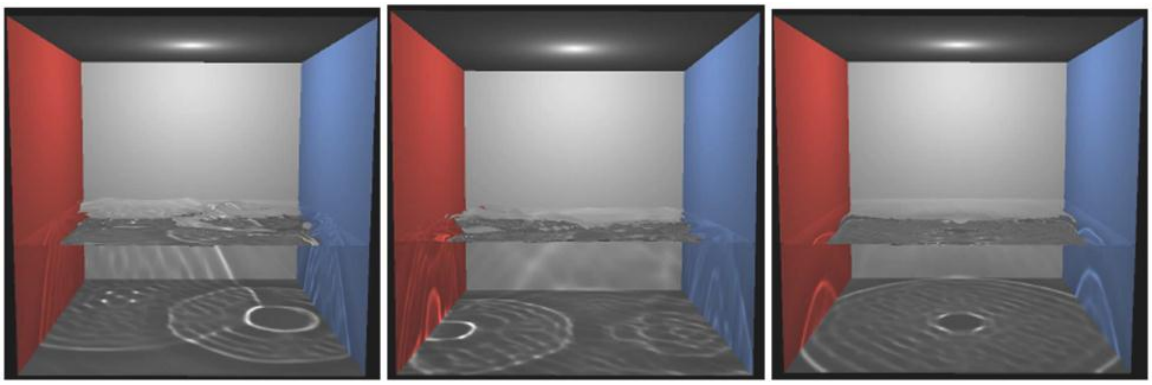
The creation of the Caustic Map requires more of an explanation. Each vertex in the vertex grid gets refracted based on the g-buffer values of the receiver geometry (that was previously rendered form the light's point of view). Using the original direction of the light, and the receiver geometry's position and normal at each pixel it is possible to calculate a refracted direction for the light. This ray is then run through an iterative intersection method. This method involves projecting points along the refracted ray into the light's view space and determining the distance to the receiver geometry located at the pixel that the projected point mapped to. This distance is used as a secondary estimate for the distance along the refracted ray where the receiver geometry might be located. The algorithm requires some mapping between various spaces, and for fewer iterations is only an approximation, but compared to doing ray intersections on all of the geometry in a scene, it is a welcome optimization that actually can converge to the correct solution.



A basic diagram of the caustic mapping algorithm. They yellow line shows where the light hits the refractive surface. r is the calculated reflected light vector. $p1$ is the first estimate of the refracted location. Projecting $p1$ to the position texture we get the white square point. The distance d' from v to the receiver geometry is used as the next estimate from v along r, resulting in $p2$ being the next estimated location of the caustic intersection with the receiver geometry. (Images from [SKP07])

Intensity of each vertex is determined based on the amount of total rays (vertices in the vertex grid) that get refracted through the object, and the surface normal at the point of refraction. The number of vertices that get refracted is determinable via occlusion querying (number of pixels that pass the Z-test). Color of each splat is determined by implementing an absorption coefficient into the material of the refracting object. A second initial pass wherein back faces are rendered allows us to determine the distance that the refracted light must travel through the refracting object.

Generally, the algorithm seems to work better on refractive objects than reflective objects. Other attempts have been made to improve upon the various shortcomings of Caustic Maps. For instance, Chris Wyman developed Hierarchical Caustic Maps to help speed up and improve some of the artifacts that result from the original caustic mapping algorithm.



Results of the caustic mapping algorithm for simulating caustics in a water environment. (Images taken from [SKP07])

4. Original Contribution

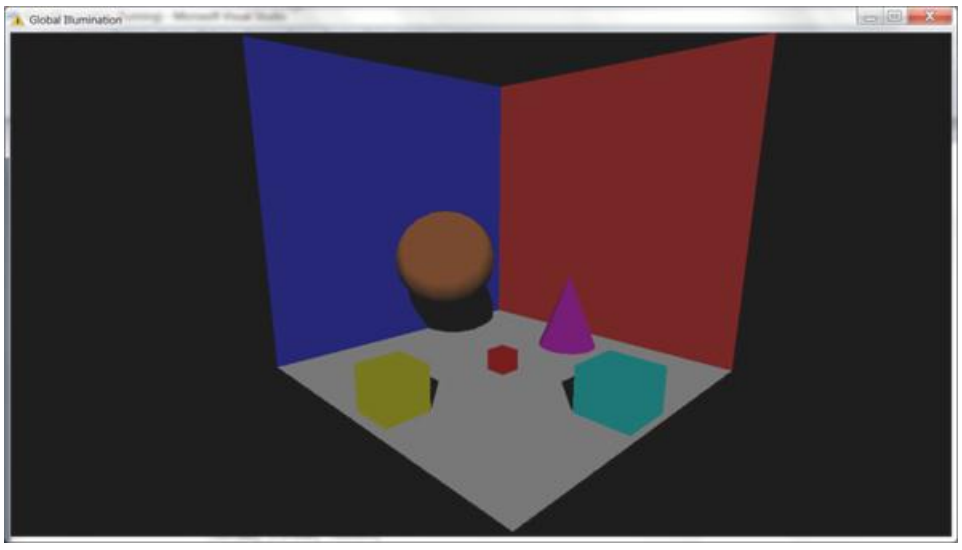
The previous section was a look into the various directions that research into Indirect Illumination has expanded. Of the presented methods, the most popular ones in practical real-time applications seem to be the Instant Radiosity methods that utilize modern GPU rasterization technology to generate VPLs and calculate indirect light. In particular, Reflective Shadow Maps and Splatting Indirect Illumination are very basic techniques that many other techniques have been built upon. However, even with modern GPU technology the base techniques can be too computationally expensive to provide good looking results at the frame rates that modern interactive applications (e.g. video games) require. To this end, some newer methods like [McGuireLuebke09] use the concepts introduced in these techniques and make something new out of them. Even though newer research is being attempted, it is worth investigating whether the base techniques can be improved to such a point that they are more usable in real-time applications.

The relevant question then becomes "How can RSM and SII be improved?" This line of inquiry led to the basic premise of this thesis. After examining existing techniques, the idea of integrating Interleaved Sampling into both RSM and SII was formulated. Interleaved Sampling was seen to be a useful addition to the Instant Radiosity techniques of Imperfect Shadow Maps and Clustered Visibility, and both RSM and SII fit the criteria for being able to use Interleaved Sampling. Recall that an algorithm is a good candidate for Interleaved Sampling if it uses many individual lights, virtual or otherwise, to light a scene. Although optimizing two techniques is an interesting task in itself, this idea of adding Interleaved Sampling to RSM and SII is even more interesting due to the fact that both algorithms are so closely related to each other, but one represents a scatter operation, and the other represents a gather operation. Also recall that RSM lights every pixel by looking up nearby texels in a reflective shadow map and treating each texel as a VPL to light the pixel in question, while SII works by selecting many texels in a reflective shadow map to be VPLs, and light the scene with those lights using screen splats. The disparate nature of the two algorithms presents an opportunity

to examine the effectiveness of Interleaved Sampling on both scatter and gather operations.

There are a couple of ways that the results of this experiment can be evaluated. First of all, one can examine how Interleaved Sampling affects the speed of the two algorithms in terms of how long it takes for the GPU to complete rendering one frame's worth of indirect illumination. Second, one can examine how Interleaved Sampling affects the accuracy of each algorithm. This refers to the degree of difference between two images: one rendered with interleaved sampling and one rendered without interleaved sampling. In this thesis, the metric chosen to measure accuracy is sum of squared differences in intensity across the entire image, a fairly commonly used image similarity metric [ZitováFlusser03]. One could argue that from a mathematical standpoint, intensity is not a good way to measure image difference, as it is possible for two pixels with vastly different hues to have the same intensity. However, in this instance, the images that are being compared have similar hues, but mostly vary in lightness. As such, it is sufficient to use intensity difference as a metric.

Next is a brief discussion of the implementation details, followed by the presentation of results.



The sample scene being used for testing.

4.1 Implementation Details

There were three main stages involved in developing the demo used to gather the results of this experiment.

1) Implementation of Reflective Shadow Maps
2) Implementation of Splatting Indirect Illumination
3) Integration of Interleaved Sampling into the Reflective Shadow Maps and Splatting Indirect Illumination algorithms.

Some general notes: All work was done using the Direct3D 10 Graphics API to interface with the GPU, an NVIDIA GeForce GT 525M. The underlying implementation uses a deferred shading approach of rendering out G-Buffers containing diffuse color, view space normal, and view-space depth to perform screen-space lighting of scenes. All images are rendered at a resolution of 1024x768.

4.1.1 Reflective Shadow Maps

The implementation of Reflective Shadow Maps does not differ greatly from that described in [DachsbacherStamminger05]. At the start of the program, a series of randomly generated texture coordinate offsets (one pair for each VPL that will be used) are generated on the CPU and stored in a texture to be sampled on the GPU during the VPL Gather pass. The texture is a 1D R16G16F texture, which means the texture coordinates are represented as 16-bit floating point numbers. Then, every frame in a first pass the RSM is generated by rendering the scene from the light's (a single directional light is used) point of view, and writing out the geometry's position, color, and normal to multiple render targets. Additionally, a normal shadow depth map is rendered for direct lighting purposes.

Once the RSM is generated, the VPL Gather pass can occur. In this pass each pixel's location is transformed to find the corresponding texel in the RSM. Then, a loop over the N-many texture coordinate offsets in the lookup texture is performed in order to obtain the position, color, and normal of each VPL that we are to use. With this information one can use the formula from section *3.3.2.1* to calculate how much light

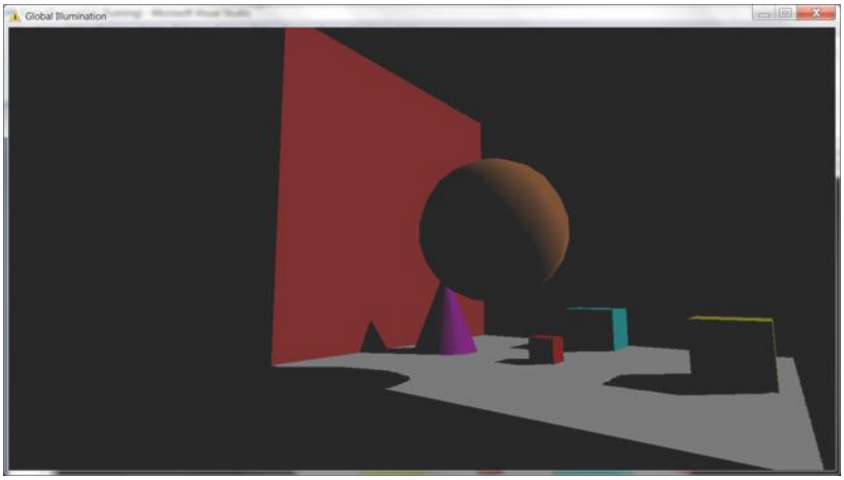
that VPL generates at our current location. This happens for every pixel in the indirect illumination buffer. As the original paper indicates, it is expensive to perform the RSM Gather at every pixel in a full-sized buffer, so the demo uses a 512x512 buffer which is upsampled using a geometry-aware blur as described in the original paper. This indirect Illumination is combined with the direct lighting calculated via deferred shading to produce the final image.

4.1.2 Splatting Indirect Illumination

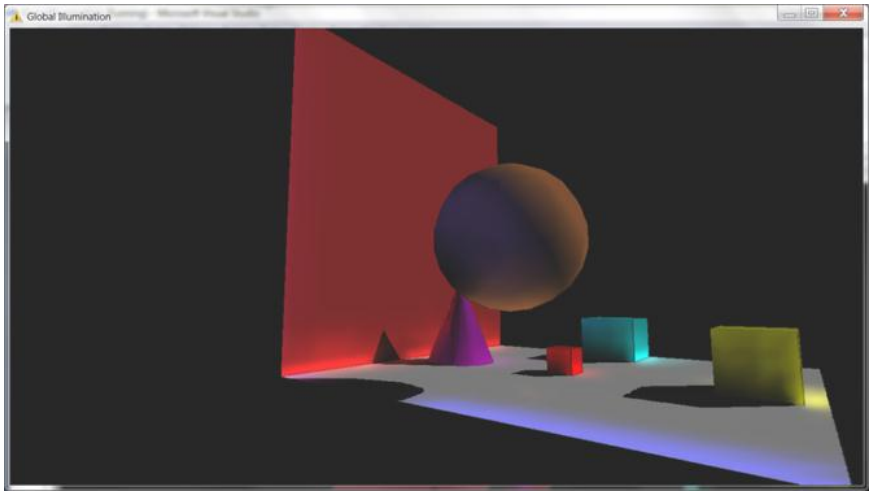
The implementation of SII starts with the generation of texture coordinates to be used as VPL sample positions in an RSM. These sample position texture coordinates are stored in a 1D R16G16F texture which is the same as how the RSM sample position offsets were stored. In this implementation, the coordinates are generated in two different ways. In the first case, a regular grid of sample positions is generated across the face of the texture. In the second case, the original sample positions can be used, but stored in a different order. The difference between the two methods will be expanded upon later. Once the VPL sample positions are generated, an RSM is rendered (generating essentially the same buffers as in the previous section), but the VPL gather pass is not performed. Instead, N-many draw calls are made, one for each VPL, each using the geometry of a sphere. In the vertex shader of each draw call, a VPL sample position is taken from the 1D texture, and the position, color, and normal of the VPL are looked up from the RSM. The sphere is then transformed to the VPL's position in projection space. This produces an effect much like deferred shading, where only the pixels in screen space that could be affected by a light are sent through the pixel shader. This process is known as splatting. For every pixel in the range of our splat, a lighting calculation is performed as if the VPL were a point light with the color of the underlying surface from the RSM.

As with the RSM algorithm, the indirect illumination via SII is performed at a lower resolution, and a geometry-aware blur is used to upsample the results. This is to help

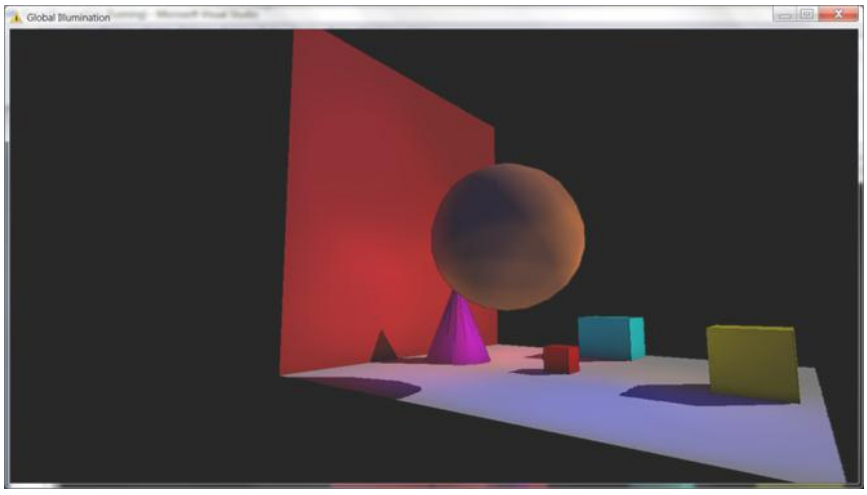
speed up the algorithm. In practice, this step is not as necessary with SII as with RSM, but it still helps.



Direct Illumination only



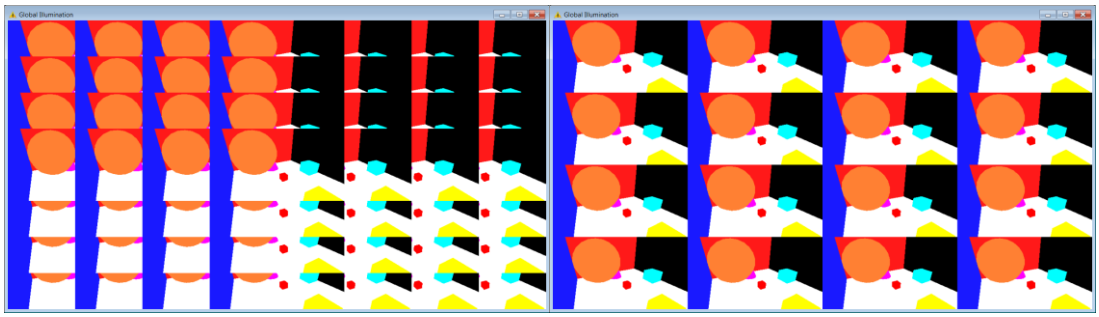
Direct Illumination + Reflective Shadow Maps



Direct Illumination + Splatting Indirect Illumination

4.1.3 Interleaved Sampling

Integrating interleaved sampling into RSM and SII was an interesting challenge. Following the advice of [SIMP06], the 2-stage swizzle process is performed in order to ensure that pixels that are part of the same sample pattern are near eachother. For a 2x2 sample pattern, the GBuffers are split up into 4 similar-looking subbuffers that each contain a unique subset of all of the pixels in the scene. This happens in a two-step approach as described in [SIMP06] by creating two lookup textures for each phase of the remapping. Each lookup texture is a R16G16F 2D texture with the same dimensions as the GBuffers being remapped. Each pixel of the lookup textures contains the texture coordinate of the texel from the original GBuffer that will be remapped to the current pixel. Then, in a pixel shader, a lookup is performed into the remap texture, and the result of that lookup is used to look up the correct texel from the original image. This required four lookup textures: two for mapping into the interleaved sample pattern, and two for mapping back to the un-interleaved pattern. At a resolution of 1280x768, this requires 15MB of memory in total. These remap textures only work for a single resolution, so if textures of multiple resolutions are being remapped, it is necessary to use a different set of remap textures for each resolution. This is the way the original paper presented the algorithm, but it could perhaps be improved and that might be an avenue for further research.



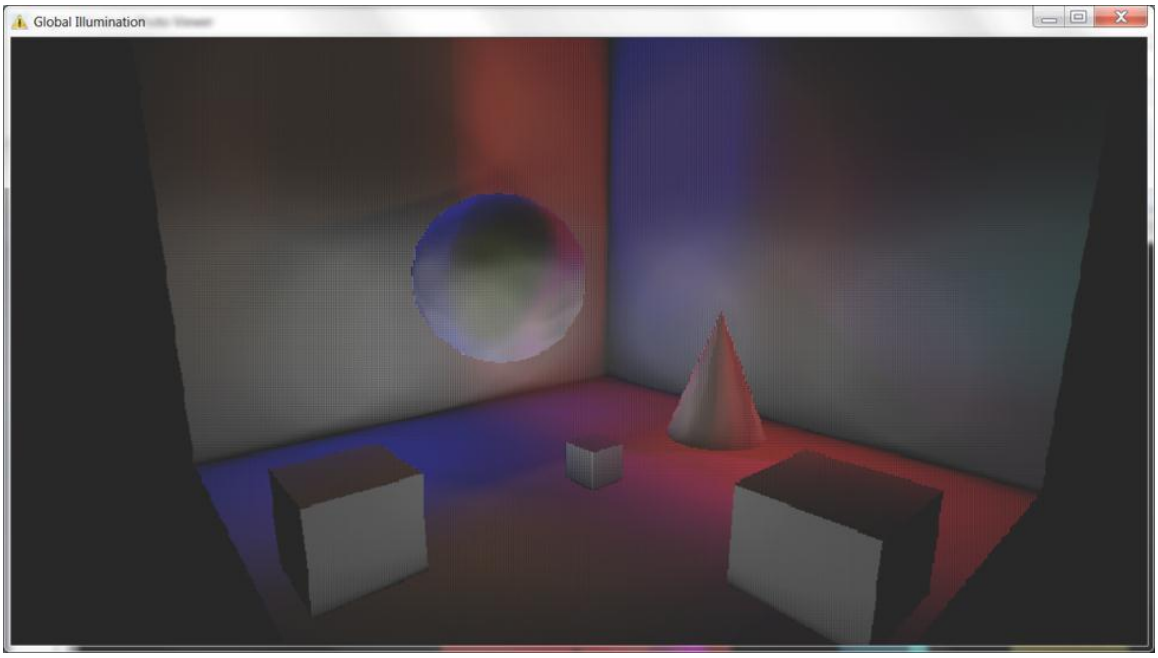
Block translation of pixels in a diffuse GBuffer into a 4x4 sample pattern.
Left: Results of the intermediate step. Right: The final buffer used for interleaved indirect illumination.

Once the GBuffers are split up into the sub blocks, indirect illumination calculations are performed. The theory is that if there are X VPLs and an NxM sample pattern, each sub-buffer will be lit with $X/(N * M)$ VPLs. For RSM, this results in performing fewer

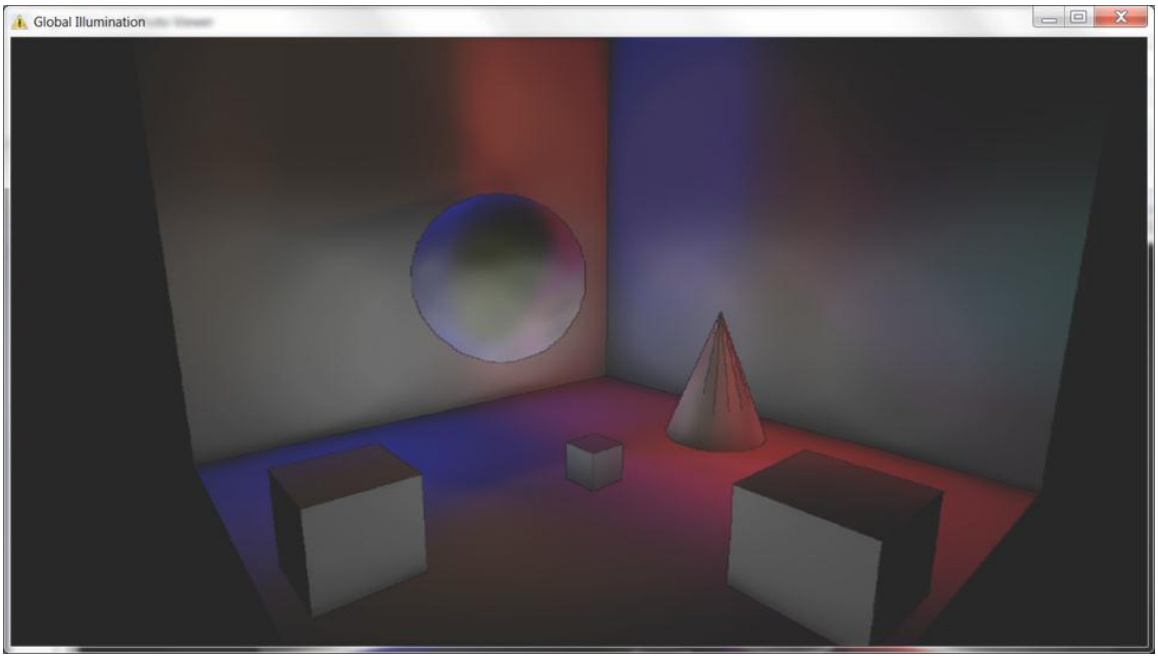
VPL calculations and hence fewer RSM texture lookups at each pixel. The VPL positions for RSM are generated randomly, so using a subset of them does not drastically alter the lighting results at every pixel. For SII, interleaving means that it is necessary to perform a subset of the splats into each individual sub-buffer. The VPL positions are usually generated in a uniform grid, and how the VPLs get distributed among each sub-buffer can make a difference in how accurate the interleaved calculations end up being. The difference between a uniform distribution of VPLs and a more randomized distribution will be explained later on.

Converting RSM and SII to use interleaved was relatively simple. It was sufficient to set the rendering viewport to the size and location of each sub-buffer, and all of the screen-space related calculations were taken care of. No special considerations were required in order to make these techniques work on the individual sub-buffers aside from specifying that fewer VPLs be used in each pass.

Once the indirect illumination was calculated, it was necessary to un-interleave the indirect illumination buffer, and perform a geometry aware blur on it. This spreads the individual contributions of each pixel to its neighbors and removes any evidence that lighting was performed with interleaved sampling. The blur used here was a separable Gaussian filter. For larger sample patterns, it is necessary to perform more blurring to make sure that the effective blur kernel ends up covering all of the pixels in a sample pattern. I.e. A single pass of a 3x3 Gaussian kernel is not sufficient for an 8x8 sample pattern. Once the blurring is done, the final results can be combined with the scene's direct illumination.



SII with Interleaved Sampling. The indirect illumination buffer displays an interleaved sample pattern.



After performing a geometry aware blur. Interleaving artifacts are eliminated.

5. Results

As previously mentioned, the effectiveness of Interleaved Sampling will be judged by measuring its effect on the speed and accuracy of each technique.

5.1 Speed measurements

Graphics programmers in real-time applications typically work with GPU time budget restrictions. As such, it is relevant to know how much of their GPU budget a certain algorithm will consume. To this end, it was chosen to measure the effect of interleaved sampling on how long it takes a GPU to complete each indirect illumination algorithm both with and without interleaved sampling integrated. A 60fps frame will allow .016 seconds for the GPU to work, whereas a 30fps frame will allow .033 seconds of GPU time. For non-interleaved algorithms, timing begins after the point of RSM generation and ends after the geometry-aware blur. For interleaved versions, the timing includes the time spent moving pixels around into the sample pattern sub-blocks, and ends after the geometry aware blur. All tests were performed at 1280x768 resolution, with RSM resolutions of 1280x768, and indirect illumination buffer size of 512x512. The RSM generation process is not included in the timings as this is the same for both interleaved and non-interleaved versions. There are four cases that are necessary to profile:

- Reflective Shadow Maps Un-Interleaved
- Reflective Shadow Maps Interleaved
- Splatting Indirect Illumination Un-Interleaved
- Splatting Indirect Illumination Interleaved

For each of these algorithms, timing data for sample sets of 64, 128, 256, 512, 768, and 1024 VPLs was collected. Additionally, for each of these configurations, 2x2, 2x4, 4x4, and 8x8 interleaved sample patterns were tested.

Num Samples	Time - RSM	Time - SII
64	0.125765	0.028453
128	0.25107	0.049206
256	0.50733	0.096885
512	0.9841	0.182759
768	1.4742	0.280525
1024	1.958312	0.362722

Timing data for *uninterleaved rendering. Time is in units of seconds.*

Num Samples	Time - RSM	Time - SII
64	0.08456	0.069322
128	0.098897	0.0427
256	0.159836	0.066594
512	0.2856	0.084725
768	0.410798	0.106727
1024	0.536653	0.132191

Timing data for 2x4 interleaved sample pattern. Time is in units of seconds.

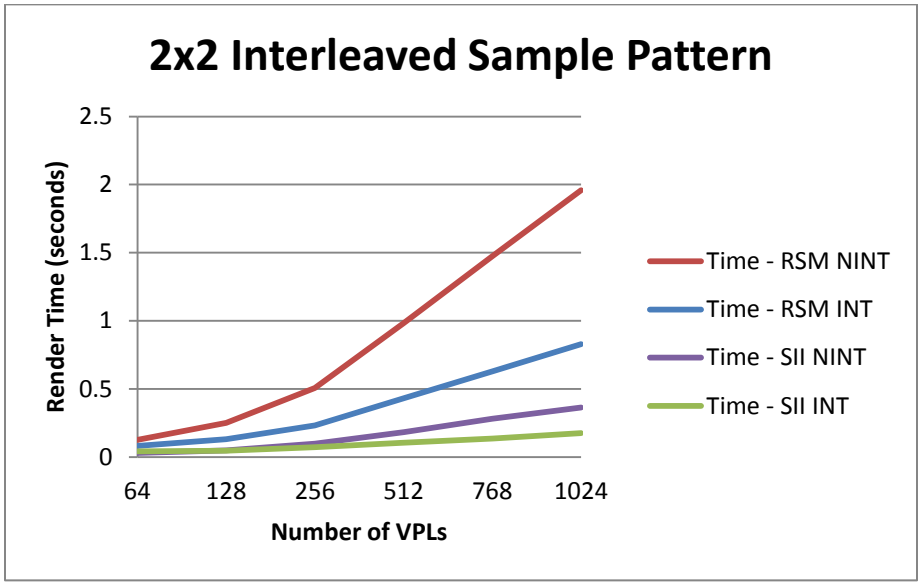
Num Samples	Time - RSM	Time - SII
64	0.051923	0.045131
128	0.079898	0.046408
256	0.117043	0.071349
512	0.191274	0.08334
768	0.265996	0.097725
1024	0.34402	0.125335

Timing data for 4x4 interleaved sample pattern. Time is in units of seconds.

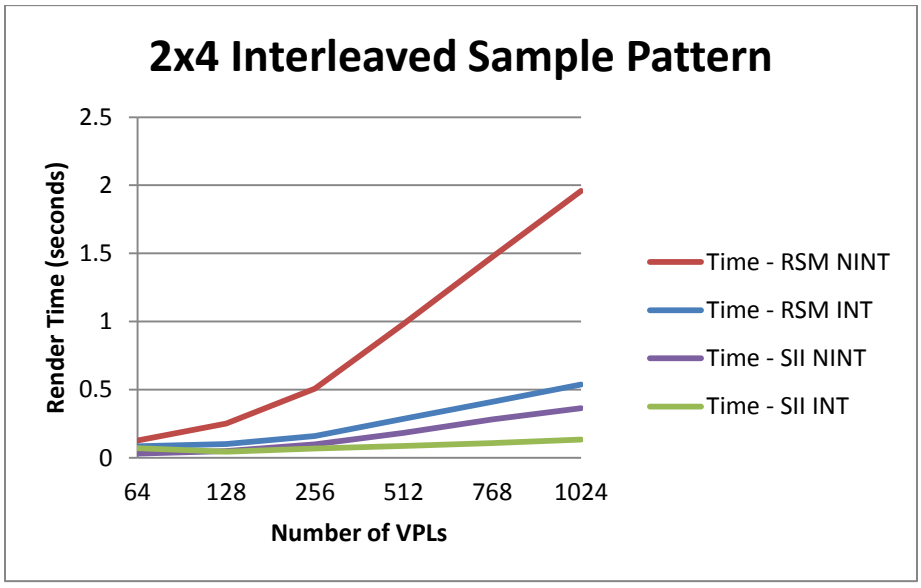
Num Samples	Time - RSM	Time - SII
64	0.055492	0.048339
128	0.061867	0.048605
256	0.073805	0.05331
512	0.099867	0.061001
768	0.125432	0.077343
1024	0.151679	0.079507

Timing data for 8x8 interleaved sample pattern. Time is in units of seconds.

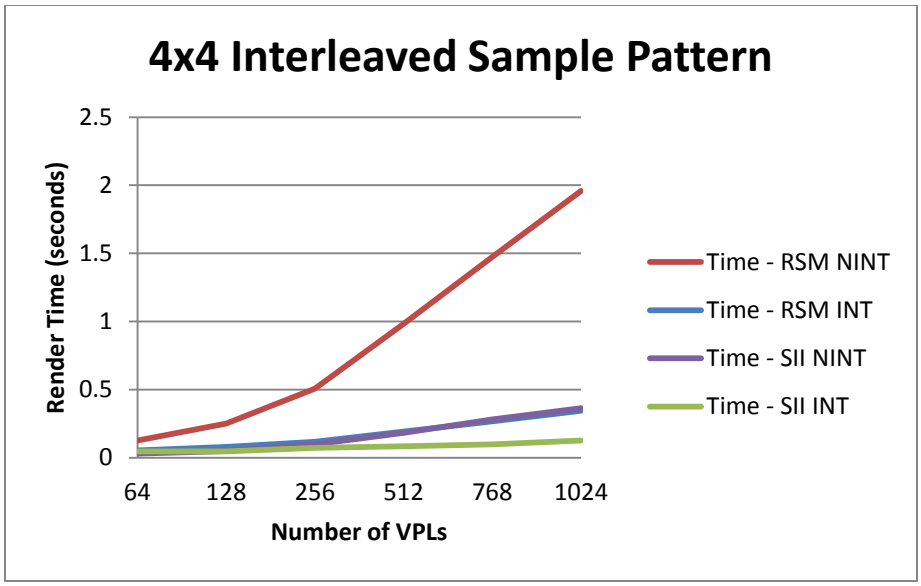
It is a bit easier to see how these tables relate to each other when the data is graphed out:



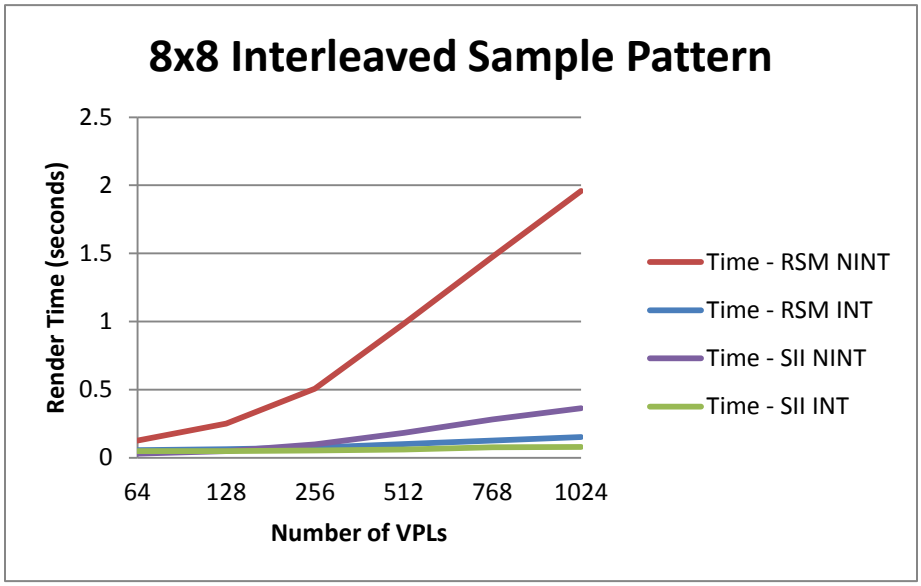
Timing data for a 2x2 sample pattern. NINT = Non-Interleaved, INT = Interleaved.



Timing data for a 2x4 sample pattern. NINT = Non-Interleaved, INT = Interleaved.



Timing data for a 4x4 sample pattern. NINT = Non-Interleaved, INT = Interleaved.



Timing data for an 8x8 sample pattern. NINT = Non-Interleaved, INT = Interleaved.

From these graphs, one can gather some interesting insights. First of all, in a non-interleaved setup, Reflective Shadow Maps is a much more expensive algorithm than SII for a given number of VPLs. Second, at low numbers of VPLs, Splatting Indirect Illumination is actually faster when not interleaved than when it is interleaved. This is due to the fact that Interleaved Sampling has some fixed overhead for swapping the

buffers around at the beginning, and swapping them back at the end. However, as the number of VPLs increase, the benefit becomes more apparent. Finally, Reflective Shadow Maps sees a much greater increase in performance than Splatting Indirect Illumination, especially as the sample pattern size increases. At 1024 VPLs, interleaved RSM takes 8% as long as its un-interleaved version. Compare this to interleaved SII, which with a sample pattern of 8x8 takes only 22% as much time as its un-interleaved version.

Intuitively, these results make sense. Interleaved sampling reduces the amount of VPL calculations that need to be performed at a single pixel. For RSM, the vast majority of work that is performed occurs in the pixel shader, and it is directly tied to how many VPLs are being calculated. Every pixel still needs to do VPL calculations, but they each have $1/(N * M)$ as many VPLs to work with for an NxM sample pattern. However, for SII the work is pretty evenly split between the vertex shader and pixel shader. Each splat has to shade $1/(N * M)$ as many pixels for an NxM sample pattern, but each pixel still needs just as much work. For RSM, interleaving also greatly reducing the amount of texture lookups that need to be performed, as this happens in the pixel shader. However, for Splatting Indirect Illumination, half of the pixel lookups happen in the vertex shader, and interleaving does not change how many times the vertex shader is fired up. Thus, RSM more greatly benefits from reducing texture lookups and ALU computations when interleaved, compared to SII.

5.2 Accuracy measurements

As mentioned previously, the accuracy of each algorithm is measured as follows: For each of the configurations that are used in the speed measurements (parameterized by the number of VPLs and interleaved sample pattern size), an interleaved frame and non-interleaved frame from the same camera location are rendered. The accuracy measurement is performed on textures containing the indirect illumination for that frame. Each image is converted to intensity, and their difference is taken. Then the sum of the square of each pixel's intensity difference is taken. The formula used to convert from RGB color space to intensity is as follows:

$$0.2989 * red + 0.5870 * green + 0.1140 * blue$$

These values take into account the fact that the human eye is more sensitive to shades of green than any other color. [Poynton97] notes that these values were better tuned to the old days of NTSC displays, and may be slightly inaccurate for modern displays. However, it still gives a value that can be used to compare the different interleaving setups that are being used.

The following tables show accuracy data for various sample pattern sizes. "RSM diff" indicates the difference between a frame of interleaved and non-interleaved Reflective Shadow Map frames, and "SII diff" shows the difference between interleaved and non-interleaved Splatting Indirect Illumination frames.

Num Samples	RSM diff	SII diff
64	1587.512573	56.348606
128	1157.870728	177.745087
256	1073.019775	4585.55127
512	972.505066	34976.08594
768	955.018005	5646.042969
1024	980.470825	57875.63281

Accuracy data for a 2x2 Sample Pattern. Values represent sums of squared intensity difference.

Num Samples	RSM diff	SII diff
64	986.992493	1263.057129
128	1364.303833	3235.810547
256	760.413147	18706.07031
512	1096.026489	75933.85938
768	1118.089722	109779.5156
1024	1027.150757	113257.8828

Accuracy data for a 2x4 Sample Pattern. Values represent sums of squared intensity difference.

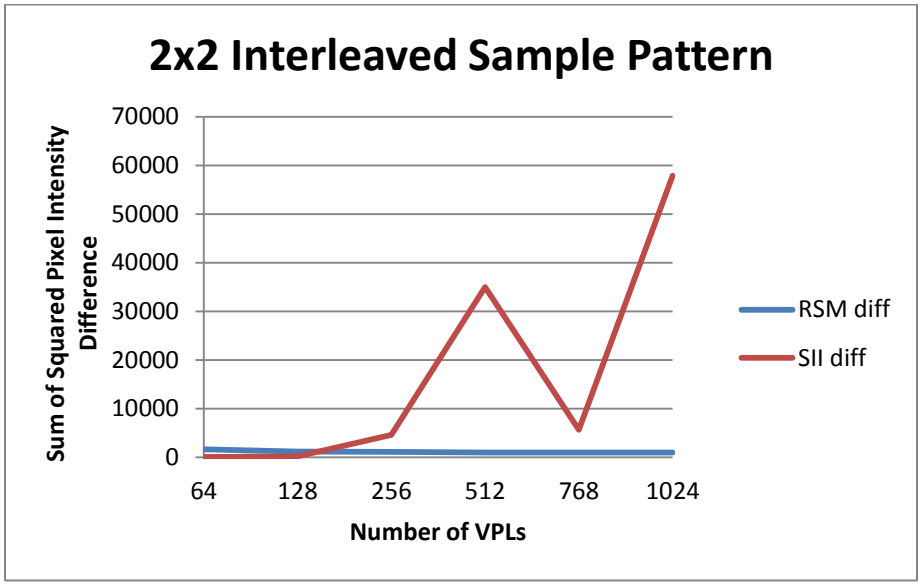
Num Samples	RSM diff	SII diff
64	2513.528564	2805.519531
128	1865.905762	7069.297363
256	1463.996704	37024.79297
512	1320.737671	111802.6875
768	1207.255981	143026.6875
1024	1163.351929	147412

Accuracy data for a 4x4 Sample Pattern. Values represent sums of squared intensity difference.

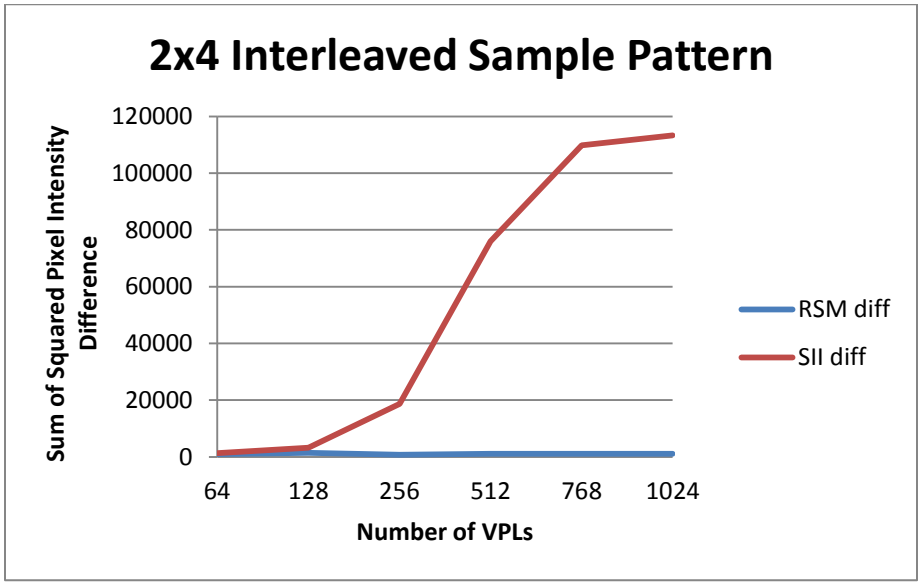
Num Samples	RSM diff	SII diff
64	8121.203125	4328.337402
128	4336.058105	10934.37305
256	3264.121338	57703.60547
512	2429.007324	171104.3594
768	2008.491577	220956.6719
1024	1650.494141	235972.6875

Accuracy data for an 8x8 Sample Pattern. Values represent sums of squared intensity difference.

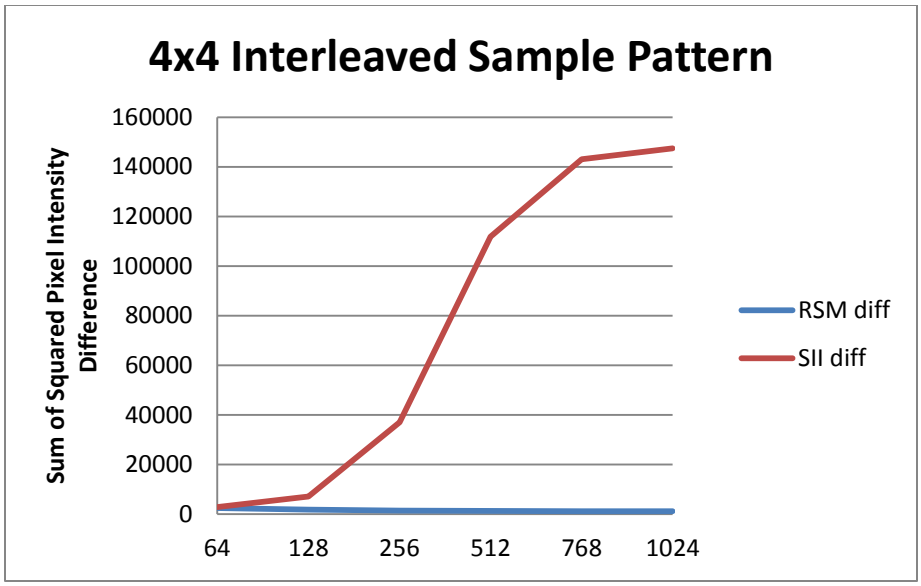
Again, it is easier to visualize this data in graphs rather than tables:



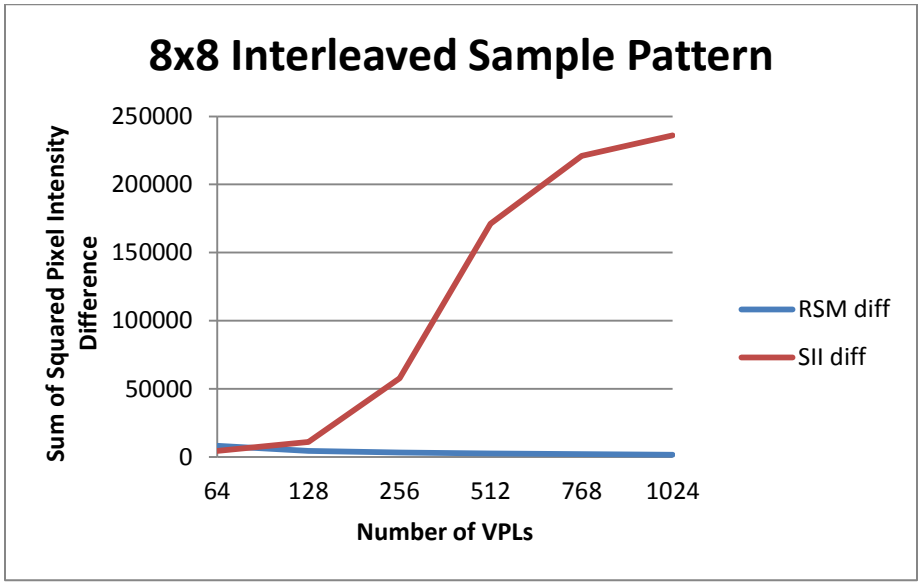
Accuracy data for a 2x2 Sample Pattern



Accuracy data for a 2x4 Sample Pattern



Accuracy data for a 4x4 Sample Pattern



Accuracy data for an 8x8 Sample Pattern

This data presents some very interesting results. First, both algorithms lose accuracy as the size of the interleaved sample pattern increases. Second, Reflective Shadow Maps tends to increase in accuracy as the number of VPLs increases. Finally, SII quickly loses accuracy as the number of VPLs increase. The first result makes sense intuitively, as fewer pixels contain accurate data and these contributions need to be spread further to

hit all of the pixels in the sample pattern, further degrading quality. The second result makes sense as well. Since we are using an interleaved sample pattern, each pixel is gathering lighting data from fewer sample sources, but as the amount of samples increases it is more likely that nearby pixels will gather similar data and we will not lose information in the blurring process. The third result requires some more explanation. Intuitively, one would think that as the number of samples increases, the results should become more accurate. However, consider the case of a 2*x*2 sample pattern. A single VPL will get rendered into only one of the four interleaved sub-buffers. None of the other sub-buffers will receive the data from that VPL, so each pixel in a 2x2 block will receive roughly ¼ the contributions of the VPL as compared to the original. This will happen for every VPL, with the error being additive across the entire image. As the sample pattern size increases the problem gets worse, as a single VPL will affect fewer and fewer pixels resulting in more and more loss of accuracy.

There is the possibility that the order of the VPLs that get splatted in interleaved SII can affect the accuracy of the algorithm. This is because placing VPLs of similar color and position in adjacent sub-buffers, rather than being in the same sub-buffer or far away in the sample pattern, prevents these pixels from receiving empty data or incoherent data from nearby pixels in the blur phase. As the sample pattern size increases though, changing the order of the samples becomes less advantageous and more similar to the original results.

6. Conclusion

In the end, it appears that Interleaved Sampling is much better suited for integration into Reflective Shadow Maps rather than Splatting Indirect Illumination. Reflective Shadow Maps sees a very sizable increase in performance since the amount of work being performed at every pixel is drastically reduced, and it does not suffer from as much loss in accuracy as Splatting Indirect Illumination does. It seems fair to extend this conclusion and say that Interleaved Sampling is friendlier to gather-based operations than scatter-based operations. This is largely due to the fact that gather operations occur at every pixel, whereas scatter operations, when applied to interleaved sub-buffers, may not. This causes a lot of information to be lost in the final blurring pass of the interleaving algorithm.

Until graphics hardware develop to the point where calculating the rendering equation at every frame becomes a possibility, graphics programmers must rely on optimizations like Interleaved Sampling to help squeeze some extra performance out of existing rendering techniques. However, judicious use of such enhancements is necessary, as it is clear that it is not suitable for all rendering algorithms.

7. References

1. [Aliaga10] Aliaga, D., 2010. Global Illumination and Radiosity. http://www.cs.purdue.edu/homes/aliaga/cs535-10/lec-radiosity.pdf (powerpoint slides)

2. [AKDS04] Annen, T., Kautz, J., Durand, F., Seidel, H.-P., 2004. Spherical Harmonic Gradients for Mid-Range Illumination. Eurographics Symposium on Rendering 2004

3. [DGRKS09] Dong, Z., Grosch, T., Ritschel, T., Kautz, J., Seidel, H.-P., 2009, Real-time Indirect Illumination with Clustered Visibility. VMV 2009

4. [DKTS07] Dong, Z., Kautz, J., Theobalt, C., Seidel, H.-P. 2007, Interactive Global Illumination Using Implicit Visibility. Pacific Graphics 2007.

5. [DachsbacherStamminger03] Dachsbacher, C., Stamminger, M. 2003. Translucent Shadow Maps. Eurographics Workshop on Rendering 2003

6. [DachsbacherStamminger05] Dachsbacher, C., Stamminger, M. 2005. Reflective Shadow Maps. I3D 2005

7. [DachsbacherStamminger06] Dachsbacher, C., Stamminger, M. 2006. Splatting Indirect Illumination. I3D 2006

8. [DSDD07] Dachsbacher, C., Stamminger M., Drettakis G., Durand F. 2007, Implicit Visibility and Antiradiance for Interactive Global Illumination. SIGGraph 2007.

9. [Evans2006] Evans, A. 2006, Fast Approximations for Global Illumination on Dynamic Scenes. SIGGRAPH 2006 – Advance Real-Time Rendering in 3D Graphics and Games, 153-171

10. [GTGB84] Goral, C., Torrance, K., Greenberg, D., Battaile, B., 1984, Modeling the Interaction of Light Between Diffuse Surfaces. Computer Graphics, 18, 3, 213-222

11. [Green03] Green, Robin. 2003. Spherical harmonic Lighting: The Gritty Details.

12. [Greger96] Greger, G., 1996. The Irradiance Volume.

13. [GrossmanDally98] Grossman, J.P., Dally, W., 1998, Point Sample Rendering. Rendering Techniques 1998, 181-192

14. [Jensen96] Jensen, H. 1996. Global Illumination using Photon Maps. Proceedings of the Seventh Eurographics Workshop on Rendering, 21-30

15. [Jensen00] Jensen, H. 2000. A Practical Guide to Global Illumination using Photon Maps.
SIGGraph 2000

16. [Jozwowski02] Jozwowski, T., 2002. Real Time Photon Mapping

17. [Kajiya86] Kajiya, J., 1986. The Rendering Equation. SIGGraph 1986

18. [Kaplanyan09] Kaplanyan, A., 2009, Light Propagation Volumes in CryEngine 3. Advances in Real-Time Rendering in 3D Graphics and Games Course – SIGGRAPH 2009

19. [KaplanyanDachsbacher10] Kaplanyan, A., Dachsbacher, C. 2010. Cascaded Light Propagation Volumes for Real-Time Indirect Illumination. I3D 2010

20. [Keller97] Keller, A. 1997, Instant Radiosity. Proceedings of SIGGraph 97, Computer Graphics Proceedings, Annual Conference Series, 49-56

21. [LSKLA07] Laine, S., Hannu, S., Kontkanen, J., Lehtinen, J., Timo, A. 2007. Incremental Instant Radiosity for Real-Time Indirect Illumination. Eurographics Symposium on Rendering 2007

22. [MKC07] Marroquim, R., Kraus, M., Cavalcanti, P.R. 2007. Efficient Point-Based Rendering Using Image Reconstruction. Eurographics Symposium on Point-Based Graphics 2007

23. [McGuireLuebke09] McGuire, M., Luebke, D. 2009. Hardware-Accelerated Global Illumination by Image Space Photon Mapping. ACM SIGGRAPH/EuroGraphics High Performance Graphics 2009

24. [NicholsWyman09] Nichols, G., Wyman, C., 2009. Multiresolution Splatting for Indirect Illumination. I3D 2009

25. [Poynton97] Poynton, C., 1997. Frequently Asked Questions About Color. www.poynton.com/ColorFAQ.html

26. [RDGK11] Ritschel, T., Dachsbacher, C., Grosch, T., Kautz, J., 2011. The State of the Art in Interactive Global Illumination

27. [RGS09] Ritschel, T., Grosch, T., Seidel, H.-P., 2009. Approximating Dynamic Global Illumination in Image Space. I3D 2009

28. [RGKSDK08] Ritschel T., Grosch T. Kim M., Seidel H.-P., Dachsbacher C., Kautz J. 2008. Imperfect Shadow Maps for Efficient Computation of Indirect Illumination. SIGGraph Asia 2008

29. [SIMP06] Segovia, B., Iehl, J.C., Mitanchey, R., Péroche, B., 2006. Non-interleaved Deferred Shading of Interleaved Sample Patterns. Graphics Hardware 2006.

30. [Sloan09] Sloan, P.-P., 2009. Stupid Spherical Harmonics (SH) Tricks

31. [SKS02] Sloan, P.-P., Kautz, J, Snyder, J., 2002. Precomputed Radiance Transfer for Real-Time Rendering in Dynamic, Low-Frequency Lighting Environments. SIGGRAPH 2002

32. [SKP07] Shah, M., Konttinen, J., Pattanaik, S. 2007. Caustics Maping: An Image-space Technique for Real-time Caustics. IEEE Transactions on Visualization and Computer Graphics 13, 2 (March/April), 272-280.

33. [WCG87] Wallace, J., Cohen, M., Greenberg, D., 1987. A Two-Pass Solution to the Rendering Equation: a Synthesis of Ray Tracing and Radiosity Methods. Computer Graphics 21, 4, 311-320

34. [Wyman08] Wyman, C. 2008. Hierarchical Caustic Maps. 2008 Symposium on Interactive 3D Graphics and Games

35. [ZitováFlusser03] Zitová, B., Flusser, J., 2003. Image registration methods: a survey. Image and Vision Computing 21 (2003) 977–1000

8. Appendices

Appendix A: Spherical Harmonics – A (very quick) Overview

(This overview is greatly based on the explanation of spherical harmonics given in [Green03])

An interesting problem in global illumination is trying to represent an entire hemisphere's worth of incoming radiance at a point on a surface. Modern computer graphics hardware is good at representing and sampling these quantities at discrete directions in the environment (for instance via ray tracing), but attempting to store an entire hemisphere's worth of information this way is both incomplete(as any attempt to represent a continuous spectrum using discrete quantities) and expensive (the amount of memory required would be enormous). As such, it is necessary to find other ways to represent hemispherical data at discrete points in an environment.

Given a continuous signal of some sort, if there is a set of functions that form a *basis* for the space that the signal resides in, it is possible to project that signal onto the set of basis functions, resulting in a set of *coefficients*. One can then use these coefficients to scale the basis functions and sum up the results, resulting an approximation of the original function. The difficulty lies in choosing a set of basis functions that allow for faithful reconstruction of the original function.

There is a subset of bases that are built up from *orthogonal* polynomials. Orthogonal bases are intriguing because if the product of any two of the functions in an orthogonal basis are integrated, the result is either 0 if they are different or a constant value if they are the same function. For *orthonormal* bases, the product will always be either 0 or 1. That is, $\int_{-1}^{1} F_m(x)F_n(x)dx$ is 1 when $n = m$ and 0 otherwise. Intuitively, these functions can be thought of as not 'overlapping' each other's influence, while still being in the same space.

Spherical Harmonics are built upon the set of polynomials known as the *Associated Legendre Polynomials*. These polynomials have two arguments l and m and are defined

over the range [-1,1] and return real numbers. There are also the set of *Ordinary Legendre Polynomials* that return complex numbers, but we are not using those here. The argument l is the *band index* of the polynomial, and is a positive integer, and m takes a value in the range $[0, l]$. The coefficients of function bands can be visualized as a triangle:

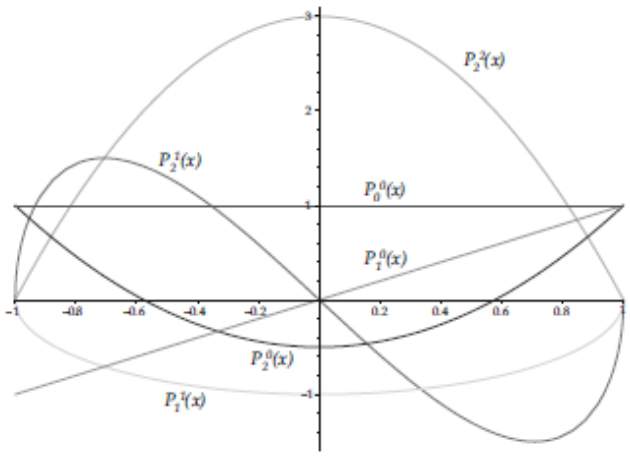
$P_0^0(x)$

$P_1^0(x)\ P_1^1(x)$

$P_2^0(x)\ P_2^1(x)\ P_2^2(x)$

…

Evaluating these polynomials is actually a quite involved process, so instead of trying to evaluate each polynomial directly, a set of recurrence relations can be used to recursively build up higher function bands from lower function bands. The curious reader should refer to an outside source for information on evaluating these recurrence relations, for instance [Green03].



A visualization of the associated Legendre polynomials. (Image taken from [Green03])

Legendre polynomials are fine for 1D functions, but they cannot be used directly on the 2D surface of a sphere. Instead, one must use Real Spherical Harmonics, which have the

associated Legendre polynomials at their core. To begin, consider the standard 2-dimensional parameterization of a sphere in 3D space:

$$(x, y, z) \rightarrow (sin\theta cos\phi, sin\theta sin\phi, cos\theta)$$

A parameterization of the spherical harmonic functions can be defined as:

$$y_l^m(\theta, \phi) = \begin{cases} \sqrt{2} K_l^m \cos(m\phi)\, P_l^m(cos\theta) \;\; when\, m > 0 \\ \sqrt{2} K_l^m \sin(-m\phi)\, P_l^{-m}(cos\theta) \;\; when\, m < 0 \\ K_l^0 P_l^0(cos\theta)\, when\, m = 0 \end{cases}$$

Where P are defined by the associated Legendre polynomials we defined above, and K is a scaling factor to normalize the functions.

$$K_l^m = \sqrt{\frac{(2l+1)}{4\pi} \frac{(l-|m|)!}{(l+|m|)!}}$$

In the case of Spherical Harmonics, l is again a positive integer, but m takes on signed integer values from $-l$ to l.

A.1 Projecting onto SH

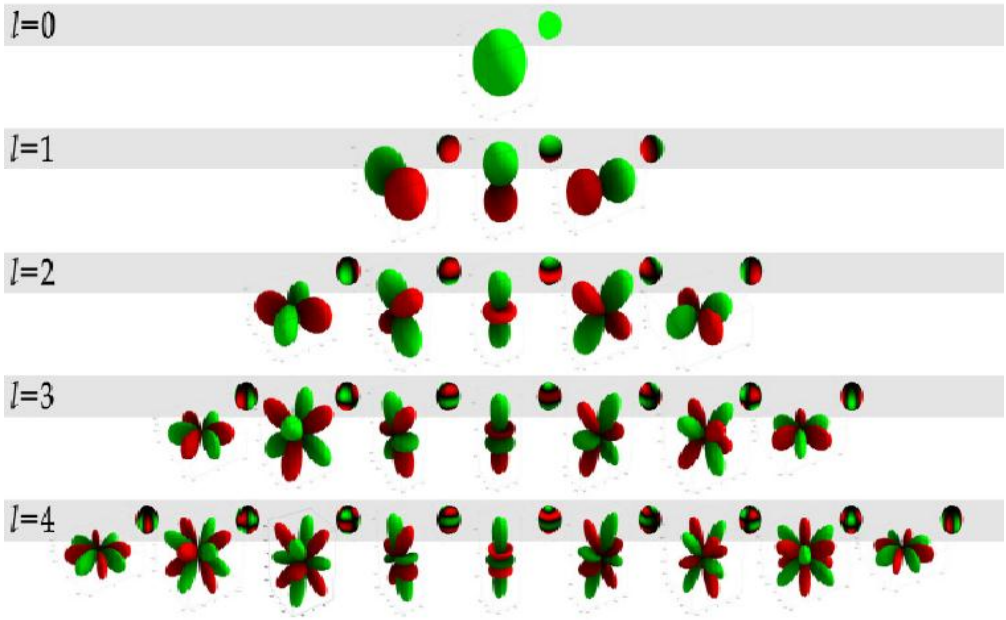
To project a function f parameterized on a sphere onto the SH basis and produce coefficients that can be later used to reconstruct the source function, it is actually rather simple. It is just necessary to calculate a single coefficient for a specific band by integrating the product of the function f and the SH function y:

$$c_l^m = \int_S f(s) y_l^m(s)\, ds$$

Note that both f and y are being evaluated at individual points. To then reconstruct the approximated function, the products of the coefficients and the spherical harmonic functions is summed:

$$\tilde{f}(s) = \sum_{l=0}^{n-1} \sum_{m=-l}^{l} c_l^m y_l^m(s)$$

As the amount of bands that are used in an approximation is increased, the fidelity with which the function can be reconstructed also increases. If the infinite series of all SH coefficients is summed up, the reconstruction would be perfect. However, in practice 4 or 9 coefficients are often used in real-time applications. This causes spherical harmonic approximations to be *band-limited*, since higher frequency features of the original signal are lost in the approximation. This is usually not a problem for representing things like global lighting and indirect illumination, as these are typically very low frequency functions.



A visualization of the spherical harmonic functions for each band. The length of the lobe in a certain direction indicates the magnitude of the function in that direction. Red lobes are negative values and green lobes are positive values. These are the basis functions for which we calculate coefficients when projecting our functions onto SH. Clearly higher bands produce more complex basis functions with more opportunity to accurately portray an input function. (Image taken from [Green03])

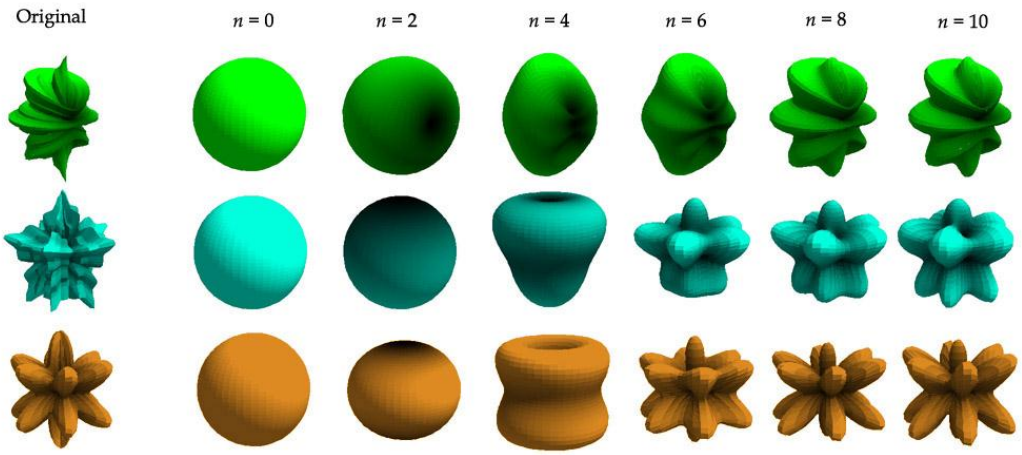
One final point about how SH coefficients are calculated needs to be addressed. How does one integrate a function over a sphere? One way to do so is via Monte Carlo Integration. If Monte Carlo Integration is used to obtain the integral of incoming light across the surface of a sphere, the incoming light from a large number of evenly distributed sample directions can be sampled and summed up, and then divided by the amount of samples that were taken. This is a trick that is made possible by the Law of Large Numbers.

The most useful part of projecting functions onto the SH basis is that if two functions are mapped to SH coefficients, the result of integrating the product of the two functions over the surface of a sphere can be calculated by simply taking the dot product of their coefficients. In lighting, one typically wants to multiply the incoming light from a direction $L(s)$ with some description of the surface reflectance (or transfer function (s)) to get the resulting reflected light. This needs to be done over a full hemisphere of light to be accurate. So if these functions are projected into SH coefficients ($\tilde{L}(s)$, and $\tilde{t}(s)$), the integral of their products can be calculated as:

$$\int_S \tilde{L}(s)\tilde{t}(s)ds = \sum_{i=0}^{n^2} L_i t_i$$

The integration of a product of two functions has now been reduced down to a simple dot product of SH coefficients.

There are many other considerations involved with using Spherical Harmonic coefficients, such as how to rotate SH coefficients, which is outside the scope of this paper, and when and how to perform Monte Carlo integration, but that's a problem that is more unique to the technique in which they are being used. The major takeaway is that projecting functions into a Spherical Harmonic basis allows us to encode a whole hemisphere's worth of information, and allows for the efficient calculation of the product of two such projected functions.



An example of how increasing the amount of bands used in projecting functions onto SH can increase the fidelity of the recreation. On the left are original functions parameterized on a sphere, and on the right are the resulting approximations for 0,2,4,6,8, and 10 bands of SH coefficients. (Image taken from [Green03])