CHALMERS

Convergent Light Volumes for Indirect Illumination

Master's Thesis in real-time indirect illumination using ray tracing and lighting volumes

JAKOB BRATTÉN

DANIEL LINDÉN

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden 2013
Master's Thesis 2013:1

Convergent light volumes for indirect illumination
Master's Thesis in real-time indirect illumination using ray tracing and light volumes

Jakob Brattén
Daniel Lindén

Examiner: Ulf Assarsson

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden 2013

Abstract

This thesis examines a technique to approximate indirect illumination in real time on modern computer hardware. Incoming light information is stored in a regular grid, called a light volume, by rasterizing large quantities of line segments into a 3D texture. Since indirect illumination is generally quite consistent between frames, several light volumes can be merged to achieve more stable lighting. In order to allow fast changes of the indirect illumination from fast moving light sources, each cell in the light volume can be weighted differently when they are merged. Since all of the light information is stored in a grid, dynamic as well as static objects can be lit using the same technique which ensures consistent lighting.

Acknowledgements

We would like to thank Avalanche Studios in Stockholm for opening their office to us. Also, Christian Nilsendahl, our supervisor on Avalanche studios who provided very good support and help with the project. Finally, we would like to thank our supervisor on Chalmers, Ulf Assarsson.

Contents

Chapter 1

Introduction

1.1 Background

Objects in the real world are lit by vast quantities of photons which are governed by complex rules of emission, reflection and refraction. When rendering computer generated images, it is not feasible to model all photons. Instead, the behaviour of light is simplified by using a wide arrange of tricks and approximations. In the most simple approximation, indirect illumination is simply a static value. While ensuring that areas not directly lit are not completely dark, this is not very accurate, and may result in flat-looking results (see Figure 1.1).

Offline rendering algorithms (Kajiya, 1986) has for a long time been able to accurately simulate indirect lighting, but the computationally heavy nature of the problem has prevented accurate real time implementations. With increasingly powerful hardware and more sophisticated algorithms, it is becoming more feasible to calculate more accurate indirect illumination in real time (Kaplanyan and Dachsbacher, 2010), (Andersson, 2011).

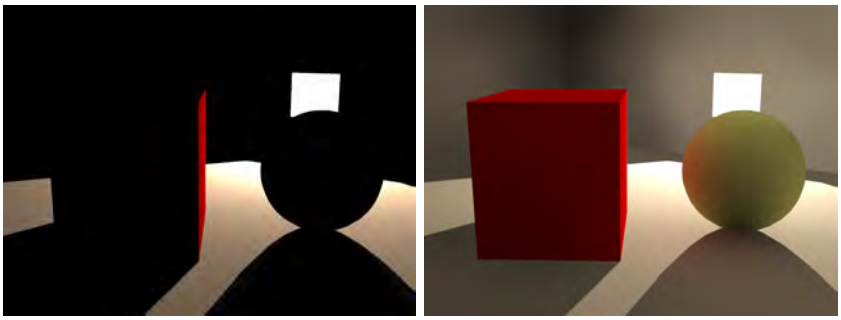
1.2 Purpose

We aim to explore a novel approach to approximate indirect lighting in real time. More specifically, the solution is aimed for game or game-like applications. This means that we need to consider dynamic geometry and dynamic light sources.

1.3 Problem

The solution is targeted at the capabilities of current PC hardware, running DirectX 11 or better. We are making use of GPGPU technology that is not available on the current consoles (Xbox 360 and Playstation 3). Consulting with our supervisor at Avalanche, we reasoned that at most 5 milliseconds per frame should be dedicated to indirect illumination in a high-end game-like application.

While we have focused on static geometry in this thesis, the algorithm is deliberately designed so that extending it to support dynamic objects is possible, and that the lighting model is consistent between static and dynamic objects.



(a) Static ambience term **(b)** Ray traced indirect illumination

Figure 1.1: A comparison between an image rendered with and without indirect illumination

1.4 Limitations

Only diffuse indirect illumination is modeled, or *LDDE* light paths. We have not examined specular light reflections in this algorithm.

Although all objects receive indirect illumination, the occlusion of the indirect light is limited to static objects. However, we have designed the algorithm so that an extension to support dynamic occluders would be possible. Though, it is not something we have examined.

In outdoor scenes, the sky is an important source of indirect illumination due to the scattering of light in the atmosphere. We will not simulate this type of indirect light source. The thesis is limited to approximating indirect lighting from point lights and directional lights.

1.5 Source code

The source code is available in a public google code repository under the MIT license at *https://code.google.com/p/lighting-thesis/*. We have been using an existing project as a base, and built our technique into this project. The code that is of interest to this algorithm is the file */source/CrazeGraphics/Renderer/LightVolumeInjector.cpp* and its associated header file, the shaders in the folder */source/CrazeGraphics/ Shaders/RayTracing*, the shader */source/CrazeGraphics/Shaders/IndirectLighting.psh* and finally a small amount of code to tie it together is present in the file */source/CrazeGraphics/ Renderer/Renderer.cpp*.

Chapter 2

Previous work

2.1 Path tracing

Classically, path tracing (Kajiya, 1986) has been used to calculate global illumination. It uses monte carlo sampling of the scene to approximate the integral of all incoming light directions for each shaded point in the scene. Because of this, images that are not using enough samples may contain noise. However, the images will have true global illumination, including high frequency phenomena such as caustics. This technique is quite slow and high quality images without noise requires offline rendering.

2.2 Photon mapping

In photon mapping (Jensen, 2001), photons are traced from the light source and stored in a data structure called a photon map where photon rays intersect with geometry. The technique is significantly faster than classic path tracing and produce images with less noise while still being capable of producing images with full global illumination. By accelerating the first bounce and the final gather using the GPU, McGuire and Luebke (2009), has created a variant of this technique that runs with interactive frame rates.

2.3 Cascaded light propagation volumes

Cascaded light propagation volumes was introduced by Kaplanyan and Dachsbacher (2010), and is a real-time technique for indirect illumination. It is based on injecting virtual point lights into a lattice and propagating the light contribution over a few iterations. Geometry is injected into a separate lattice in order to support occlusion of the indirect light. The technique is very fast and has been implemented in real time on current generation consoles (Microsoft XBOX 360 and Sony Playstation 3), but discretizes the light and thus reduces accuracy.

2.4 Radiosity

Radiosity (Goral et al., 1984) precomputes the form factors between all pairs of surfaces. A form factor describes the amount of energy transferred by diffusely reflected light between two surfaces. In order to light a surface, the scene is iterated to propagate the energy from surfaces with direct lighting and thus achieve indirect illumination.

2.5 Instant radiosity

Keller (1997) introduces instant radiosity, which is an algorithm that spawns light sources in the scene. These lights are used to approximate the indirect illumination. The author intended the technique to be used to calculate the illumination of a single image. However, using a modern deferred shading (Deering et al., 1988) rendering pipline that is capable of rendering hundreds of lights every frame (Andersson, 2011), this technique could be implemented in real time.

2.6 Precomputed radiance transfer

Precomputed radiance transfer (Sloan et al., 2002) uses precomputed transfer vectors to calculate the indirect illumination in a scene. These transfer vectors are stored as spherical harmonics for the vertices of the scene. When lighting the scene, the light is converted to a spherical harmonic approximation, and the amount of light affecting each vertex can then be calculated with the precomputed transfer vectors. The transfer vectors are independent of incoming light, and thus the light sources can be dynamic. However, the precomputed transfer vectors rely on the geometry being static.

2.7 Voxel cone tracing

By storing a voxelized version of the scene with direct lighting information, Crassin et al. (2011) renders both diffuse and specular indirect illumination in real time with cone tracing. In order to save memory and speed up the cone tracing, a sparse voxel octree is used. The voxelization of static geometry can be precomputed once, and voxelized dynamic objects can be merged with this representation. The algorithm supports fully dynamic lighting as well, because the direct lighting is rasterized into the sparse voxel octree, which is then used when cone tracing. By using a few cones with wide angles, diffuse indirect illumination is calculated, and a cone with a narrow angle is used to calculate the specular indirect illumination.

2.8 Screen space techniques

By sampling the scene in screen space, techniques such as screen space directional occlusion (Ritschel et al., 2009) can provide plausible short-range indirect illumination. A simpler version of this algorithm is screen space ambient occlusion (Mittring, 2007), but it does not provide any color bleeding. An advantage

of this kind of algorithms is that since they operate in screen space they are very predictable and stable in performance. Also, performance scale independently with the geometric complexity of the scene.

Chapter 3

Algorithm

3.1 Overview

First, we generate a set of rays approximating a small amount of photons after the first bounce of the light. We do this for every light source in the scene contributing to the indirect illumination. Intersection tests are performed between each of these rays and the scene geometry, and rays are cropped to stop at their first intersection point with geometry if such point exists. After this, the lighting that these rays represent are injected into a 3D grid structure. From this 3D grid, we calculate the amount of indirect lighting in the final rendered image.

3.2 First bounce and the reflective shadow map

In order to quickly calculate the first bounce of the rays we use a *reflective shadow map*, RSM, (Dachsbacher and Stamminger, 2005). The RSM is a texture data structure that contains information about position, albedo and normal for each pixel in a viewport. It is very fast to create by simply rasterizing the scene. As shown by McGuire and Luebke (2009), by rasterizing the RSM from the viewport of each light source, we can interpret every pixel in the RSM as an intersection between the light of the light source and the geometry of the scene. These intersection points can then be used to calculate reflected rays by randoming a ray in the hemisphere of the point. In order to cull away unnecessary rays, an intersection test between the ray and the view frustum is performed. If the ray contributes lighting to the visible scene, the ray, along with the albedo of the pixel it is reflected from, is stored. Rays also contain information about the light source they are originating from: the color of the light and the dynamicity (see Section 3.4.5) of the light source. The reflected rays generated in a frame can be seen in Figure 3.1.

In order to focus computational power on the rays that contribute the most, we use *russian roulette* (Arvo and Kirk, 1990). Russian roulette is used to discard rays with low intensity, so that low brightness rays are more rarely processed. The brightness of all rays are then scaled to the inverse of the survival probability, resulting in a smaller set of rays with less diversity in brightness. The result is that the expected value of the lighting remains unchanged, but the

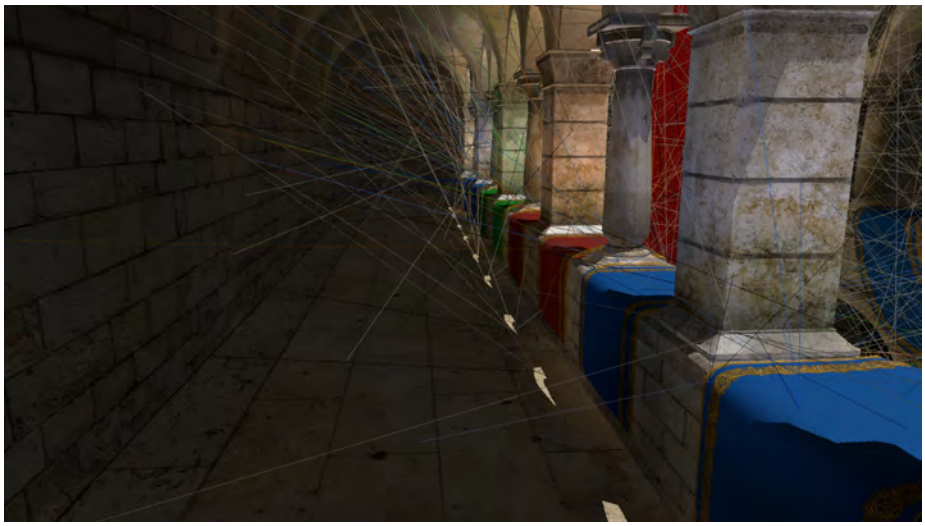


Figure 3.1: Intersection rays are rasterized for demonstration purposes. Note how rays originate from directly lit areas and how each ray's color is determined by the scene material.

variance increases. Russian roulette increases the noise of the indirect illumination in the final image. However, since we are merging light volumes, the noise is less noticeable.

3.3 Ray intersection testing

The set of reflected rays are intersection tested against a low resolution version of the scene geometry. Each ray is then converted to a line segment (though in this report, it will still be referred to as a ray), that ends at the first intersection point between the ray and scene geometry.

By performing intersection tests against a greatly simplified scene, performance is improved. The indirect illumination is generally of quite low frequency, and details in the intersection testing will not be visible in the final image due to the low resolution light volume. An artist could construct a less complex approximation of the original scene manually, or the process could be automated by using some kind of mesh optimization algorithm, *e.g.* the one presented by Hoppe (1999).

3.3.1 Spatial data structures

A spatial data structure is used for fast ray intersection testing, such as the *kd-tree* (Bentley, 1975). Kd-trees for the geometry can be built at runtime (Wald and Havran, 2006). An algorithm for fast tracing of rays against this acceleration structure is necessary, such as the one presented by Popov et al. (2007), which is intended to be implemented on the GPU. By implementing the ray intersection testing on the GPU, copying the data between the CPU and the GPU back and forth can be avoided.

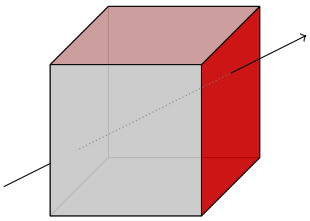


Figure 3.2: A light volume cell is most easily visualized as a cube whose side colors corresponds to the radiance inside the cube. Here, a red ray intersects the cell and the cube's sides are colored based on the ray's orientation and color.

Figure 3.3: Artifacts from using spherical harmonics to store the light in the light volume is visible as large black spots on the floor.

3.4 Storing the light

3.4.1 Light volume data structure

The lighting of the scene is stored in a three dimensional grid. We call this data structure a *light volume*. Every cell stores information about the incoming indirect illumination. By using a grid, dynamic objects can be lit by using the same technique as static objects, unifying the appearance of all objects in the scene. Each cell in this grid stores incoming light from all six sides of a cube, similarly to the ambient cube technique presented by Mitchell et al. (2006) or a cubemap with 1x1 side resolution (see Figure 3.2). Additionally, we also store the dynamicity (see Section 3.4.5) of the light for each side of the light volume cell. This storage technique requires six textures with four channels: three channels for the color of the lighting and one for the dynamicity.

It is also possible to store the lighting in a cell using spherical harmonics. We tried this, but we noticed some artifacts (see Figure 3.3) under certain conditions. The problem we faced was that spherical harmonics that were created in a cell where a lot of rays were traveling in a roughly similar direction often caused strong negative lighting in the opposite direction. However, when using

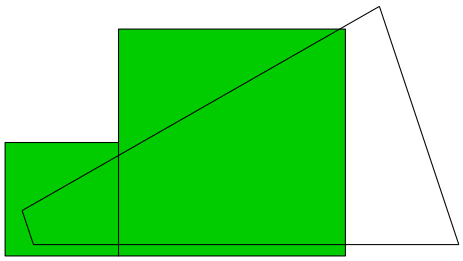


Figure 3.4: A 2D projection demonstrating the light volume positioning. The colored boxes are axis aligned bounding boxes, encapsulating the view frustum.

spherical harmonics, it is possible to reduce the memory requirements of the light volume. Storing the first two orders of a spherical harmonics requires four values, resulting in a total of three RGBA textures plus an additional single channel texture to store the dynamicity.

3.4.2 Light volume positioning

Having a static light volume covering the entire scene is not feasible for anything but very small scenes. Instead, we use dynamic light volumes that covers a subsection of the view frustum. As suggested by Kaplanyan and Dachsbacher (2010), the light volume is positioned in discrete steps the size of a cell. This removes some temporal light shimmering artifacts that occur when the light volume moves and also enables us to merge the indirect lighting results between frames (see Section 3.4.5).

Another option is to position the light volume with the camera in the center. However, this reduces the effective resolution of the light volume since such a large part of the light volume would exist outside of the view frustum. On the other hand, fast camera rotation would enable reuse of more lighting information from previous frames.

3.4.3 Cascades

To further increase the effective light volume resolution, we propose using cascaded light volumes. Further away from the camera, the quality of the indirect illumination is less important. By adding additional light volumes with increasingly large cells behind the primary light volume, we can store indirect lighting at great ranges while retaining high light volume resolution near the camera. If a scene expands beyond the last light volume, we simply extrapolate the lighting information by clamping the light volume texture.

3.4.4 Injecting the rays into the light volume

The rays are injected into the light volume, which is a 3D texture, by rasterizing them as lines. Since the whole ray is injected into the light volume, instead of just the beginning and the end of the ray, any objects along the path of a ray will receive indirect illumination as well. Dynamic objects, even if not included in the geometry of the kd-tree, will therefore be lit as well.

The lines could be rasterized with hardware accelerated line antialiasing, which will effectively convert each ray into a cone instead. This improves the stability of the lighting as the result is more blurred. However, this technique resulted in significant light bleeding through geometry.

Tessellating the rays

When rendering geometry into a 3D texture, special care has to be taken if the geometry is to be rasterized into more than one z-slice of the texture. In Direct3D 11, a single primitive can only be rasterized into a single z-slice at the time. The z-slice it is rendered into is controlled by the *RenderTargetArrayIndex* property set in the geometry shader. Since a ray is likely to intersect with several z-slices, the ray needs to be split at these intersection points.

First, we tried doing this using the tessellation pipeline available in Direct3D 11. However, we also tried tessellating the rays in a compute shader, and then store the result into a new buffer. The latter technique proved to be slightly faster. Performing tessellation in a compute shader also makes it possible to tessellate the lines into an arbitrary amount of subdivisions. The tessellation unit is limited to 64 segments (Mic, 2012b), essentially limiting the resolution of the light volume to 64x64x64 without having to clip some lines intersecting a lot of z-slices.

The input to the compute shader tessellator is a buffer containing all rays. The rays are processed and the tessellated segments are stored in a AppendConsumeStructuredBuffer. At first we only cloned the rays so that there were enough rays for all the slices, but by instead clipping each segment to only be as long as necessary we noted a significant performance gain. This clipping is performed by iterating over all of the segments, finding the next intersection point for each segment in a spirit similar to Amanatides and Woo (1987).

Rasterizing the rays

In Direct3D 11, the rasterization rules (Mic, 2012a) states that a pixel of a line is drawn if the line exits a diamond shape in the pixel. By reversing the line segments, so they start where the ray ends, the pixel where the ray ends is more likely to be drawn and the pixel where the ray starts is less likely to be drawn. This helps reducing self-illumination, which is a problem with storing lighting in a grid, while ensuring that the ray illuminates its intersection point with the geometry.

In order to calculate the amount of light a ray contributes to a side of a cell in the light volume, we use $\mathbf{L}_i = max(0, \mathbf{n} \cdot \mathbf{l}) * \mathbf{c}$. Where $\mathbf{L}_i$ is the amount of incoming light to a side in the cell, $\mathbf{n}$ is the normal of the side, $\mathbf{l}$ is the direction of the light and $\mathbf{c}$ is the color of the light. By using additive blending, the total amount of incoming light for every side in every cell is accumulated.

3.4.5 Frame merging

In order to reduce the number of rays that needs to be processed each frame, we use old light data as stored in the light volume and blend the current frame's lighting information with the history of previous frames. Since the light volume is positioned around the view frustum, spinning the camera will reposition the

light volume in world space. When a light volume repositioning occurs, the content of each cell is moved so that the light data remains stable in world space. In such a repositioning, there are always some cells at the edge of the volume that moves into a world position previously unoccupied by the light volume. These cells has no information of how light behaved at this world position at the previous frame. In such cases, we preserve the light data we had before the move as a first guess of the lighting value. Further, we increase the dynamicity (see Section 3.4.5) of each ray rendered in this cell for a short duration in order for the cell to more quickly converge into a more accurate result. Since the light volume is essentially an axis aligned bounding box surrounding the view frustum, there are in many cases some buffer space between where the frustum ends and the volume ends. Because of this, only very fast camera movement will enable cells that are not fully converged to appear in the view frustum.

Dynamicity If the contribution of the rays rasterized each frame is too high, the indirect light becomes unstable and will appear to flicker or pulsate. If the contribution is instead too low, a moving light's indirect illumination will origin from positions that are no longer directly lit. This would effectively cause the indirect light to unnaturally lag behind the dynamic direct light. To counter these problems, we introduce a concept called dynamicity. In addition to color and direction, rays inherit the dynamicity value of the light source. A light source's dynamicity is determined by its movement speed and, if applicable, its rotational speed. When the rays are rasterized into the light volume, the highest dynamicity off all the different rays intersecting the cell is stored in that cell. We do this by using the max blend mode for the dynamicity color channel in the graphics hardware. This aggregated dynamicity value is then used as alpha when blending the light volumes, a higher dynamicity shifts more weight to the new light volume.

3.5 Rendering

The light volume has a known world position and it is a simple matter to find a pixel's corresponding cell in the light volume. We compare the normal of each pixel to the irradiance of its corresponding light volume cell and use this to evaluate the contribution of the indirect lighting for the pixel.

We add this direction dependent indirect lighting term to the pre-existing lighting model as ambience. For pixels with no indirect lighting contribution above a set threshold, we complement the ambiance with a static term. This ensures that no area is completely dark, even in the cases where sections of the light volume contain little or no indirect light, while still preserving the dynamic appearance of the indirect illumination for the scene.

Chapter 4

Results

The problem of this thesis was to approximate indirect illumination in real time, mainly for a game-like application. The algorithm we came up with has some interesting properties for such applications, such as the high image quality and good performance. However, there are several problems with it; these problems makes the algorithm unsuitable for very fast camera movement. Unfortunately, fast camera movement is a common occurrence in games.

4.1 Image quality

When compared to reference images rendered using a ray tracer, we believe that the images generated by our algorithm produce indirect lighting that correspond fairly well with ground truth.

4.2 Performance

For all performance tests, we have used an Intel Core i7-2600K CPU (3.40GHz) processor and a single Nvidia GeForce GTX 560 Ti graphics card. As described in the problem statement, the performance requirement of this algorithm was that it had to run faster than 5 milliseconds per frame. Otherwise, it would not be viable for real time use in computer games.

We have tested the algorithm using two different settings on the resolution of the light volume: 16x16x16 and 32x32x32 pixels. Both of these resolutions performed within our performance target. The 16x16x16 quality setting ran the sponza scene in 3 milliseconds, while the 32x32x32 quality took 4 milliseconds. In Figure 4.3, we show images rendered an image using two different quality settings and the timings we measured.

The step of the algorithm that takes the longest to process is to rasterize the rays into the light volume. The timings vary depending on how many rays that are in the frame, but the timings for an image where most of the scene is in view can be found in Table 4.1.

(a) Ray tracer reference render

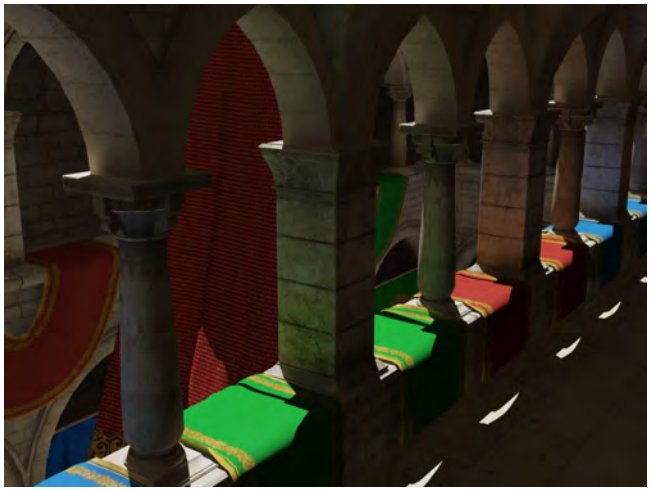
(b) Our algorithm, using 32 cubed light volume resolution

Figure 4.1: The Crytek sponza atrium scene, using a ray tracer and our algorithm respectively.

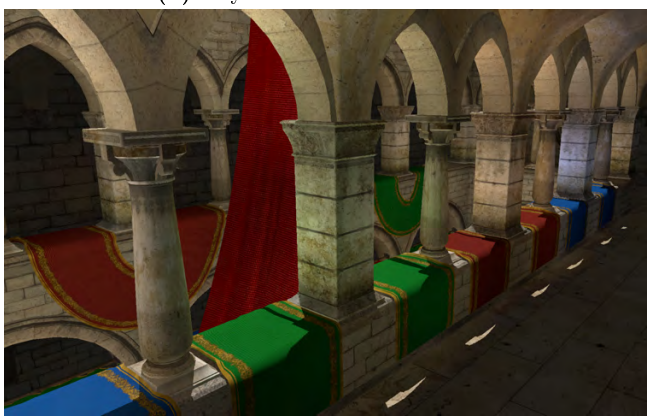
4.3 Problems

In certain cases light may appear to bleed through opaque geometry and appear on geometry closely behind it. This is due to to the discrete nature of the light volume, and the problem can be remedied by increasing the resolution of the light volume.

Additionally, we experience temporal instability, or shimmering. We introduced the concept of dynamicity to counter this phenomena, but there are still times, especially when the camera or the light sources moves very slowly, when this shimmering is noticeable. On the other hand, if the algorithm is tweaked incorrectly so that the light dynamicity is too low, there will be notable delays when the light volume converges to its correct value.



(a) Ray tracer reference render



(b) Our algorithm, using 32 cubed light volume resolution

Figure 4.2: An alternative camera angle, presenting the iconic light bleeding effect of indirect lighting.

Step of algorithm	Time (in milliseconds)
Rasterizing rays into light volume	1.7
Ray intersection test	1.0
Apply indirect lighting	0.7
Generating rays	0.5

Table 4.1: Performance per step of the algorithm in a typical frame of the sponza atrium scene.

(a) 16x16x16 light volume (2.7 milliseconds)

(b) 32x32x32 light volume (3.9 milliseconds)

Figure 4.3: A test scene rendered using different settings.

Chapter 5

Discussion

We are pretty happy with the quality of the lighting that the algorithm produces. However, the problems with the instability and the short time it takes for the lighting volume to converge, makes it unsuitable for most types of games. If these problems were to be solved though, the algorithm does not require much extra work from artists in order to be implemented in a game. The only new geometry required is the one used for ray intersection testing. This geometry could, as previously noted, be generated from the game assets. Alternatively, if the game has a physics engine, it could be possible to use the physic engines shapes directly. Such physics shapes are often spheres and similar mathematically simple objects. Using these shapes instead of triangle meshes could speed up the ray intersection testing step.

Due to the real time nature of the algorithm, it seems that light propagation volumes is a good algorithm to compare it to. The way that light propagates in LPV limits the distance that light can travel. However, since our algorithm is based on rays, we have no such problem. On the other hand, due to the converging nature of our algorithm, and the way old lighting information is reused, we have more problems with stability of the lighting that does not exist in LPV.

This technique does not handle high frequency light phenomena, such as detailed ambient occlusion. An ambient occlusion algorithm such as SSAO (Mittring, 2007), should be used to achieve better details in the indirect illumination.

Depending on the application in which this algorithm is used, some of the artifacts could be reworked as features. By letting each computed frame of rays have a high impact in the interpolated result, the lighting becomes unstable. This could be directly beneficial in a scene with lights such as torches, and one could indeed enforce this behaviour for certain lights. Further, by decreasing the light volume resolution, we gain a spotty, unstable result that could be used as caustics in an underwater scene.

Chapter 6

Future work

Currently, we have not examined how to handle area light sources, which, in some cases, can be very important. An example of an important area light source is the sky in outdoor scenes. The ambient lighting originating from the atmosphere as it reflects light from the sun is a significant part of the indirect lighting outside, and that is not taken into consideration at all by our algorithm. We have reasonably fast ray intersection testing which might be used to calculate the sky illumination. This is however not something we have looked into deeply and further reasearch is required.

Certain complex volumetric objects, such as the foliage in a tree, is problematic to light due to very detailed geometry. Instead of building a complex collision mesh one could use a very simple mesh and assign it a density factor. Every ray would then, based on this density factor, randomly decide to pass through the object or to treat it like a traditional object and bounce off it.

To include specular indirect lighting, the BRDF of the material could be taken into consideration in order to determine the first bounce direction. This would enable modeling of specular light bounces. Calculating specular reflections for the final bounce is, however, much more problematic.

We have not examined dynamic objects closely, but there exist techniques to, at runtime, create acceleration structures of dynamic objects on the GPU quite quickly (Garanzha et al., 2011; Wu et al., 2011). These could be run several times per second provided that the geometrical resolution of the dynamic object occlusion meshes is low enough resolution. Another possibility is to approximate the dynamic objects with primitive shapes such as boxes and spheres, which would accelerate the ray intersection testing and reduce the need for an advanced acceleration structure.

Bibliography

John Amanatides and Andrew Woo. A fast voxel traversal algorithm for ray tracing. In *In Eurographics '87*, pages 3–10, 1987. Cited on page 10.

Johan Andersson. DirectX 11 rendering in battlefield 3, March 2011. URL `http://publications.dice.se/attachments/GDC11_DX11inBF3_Public.pdf`. Cited on pages 1 and 4.

James Arvo and David Kirk. Particle transport and image synthesis. In *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '90, pages 63–66, New York, NY, USA, 1990. ACM. ISBN 0-89791-344-2. doi: 10.1145/97879.97886. URL `http://doi.acm.org/10.1145/97879.97886`. Cited on page 6.

Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975. ISSN 0001-0782. doi: 10.1145/361002.361007. URL `http://doi.acm.org/10.1145/361002.361007`. Cited on page 7.

Cyril Crassin, Fabrice Neyret, Miguel Sainz, Simon Green, and Elmar Eisemann. Interactive indirect illumination using voxel cone tracing. *Computer Graphics Forum (Proceedings of Pacific Graphics 2011)*, 30(7), sep 2011. URL `http://maverick.inria.fr/Publications/2011/CNSGE11b`. Cited on page 4.

Carsten Dachsbacher and Marc Stamminger. Reflective shadow maps. In *Proceedings of the 2005 symposium on Interactive 3D graphics and games*, I3D '05, pages 203–231, New York, NY, USA, 2005. ACM. ISBN 1-59593-013-2. doi: 10.1145/1053427.1053460. URL `http://doi.acm.org/10.1145/1053427.1053460`. Cited on page 6.

Michael Deering, Stephanie Winner, Bic Schediwy, Chris Duffy, and Neil Hunt. The triangle processor and normal vector shader: a VLSI system for high performance graphics. In *Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '88, pages 21–30, New York, NY, USA, 1988. ACM. ISBN 0-89791-275-6. doi: 10.1145/54852.378468. URL `http://doi.acm.org/10.1145/54852.378468`. Cited on page 4.

Kirill Garanzha, Simon Premoze, Alexander Bely, and Vladimir Galaktionov. Grid-based sah bvh construction on a gpu. *Vis. Comput.*, 27(6-8):697–706,

June 2011. ISSN 0178-2789. doi: 10.1007/s00371-011-0593-8. URL http://dx.doi.org/10.1007/s00371-011-0593-8. Cited on page 17.

Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Bennett Battaile. Modeling the interaction of light between diffuse surfaces. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '84, pages 213–222, New York, NY, USA, 1984. ACM. ISBN 0-89791-138-5. doi: 10.1145/800031.808601. URL http://doi.acm.org/10.1145/800031.808601. Cited on page 4.

Hugues Hoppe. New quadric metric for simplifying meshes with appearance attributes. In *Proceedings of the 10th IEEE Visualization 1999 Conference (VIS '99)*, VISUALIZATION '99, pages –, Washington, DC, USA, 1999. IEEE Computer Society. ISBN 0-7803-5897-X. URL http://dl.acm.org/citation.cfm?id=832273.834119. Cited on page 7.

Henrik Wann Jensen. *Realistic Image Synthesis Using Photon Mapping, 2nd Edition*. A K Peters/CRC Press, 2nd revised edition edition, July 2001. ISBN 1568811470. Cited on page 3.

James T. Kajiya. The rendering equation. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '86, pages 143–150, New York, NY, USA, 1986. ACM. ISBN 0-89791-196-2. doi: 10.1145/15922.15902. URL http://doi.acm.org/10.1145/15922.15902. Cited on pages 1 and 3.

Anton Kaplanyan and Carsten Dachsbacher. Cascaded light propagation volumes for real-time indirect illumination. In *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, I3D '10, pages 99–107, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-939-8. doi: 10.1145/1730804.1730821. URL http://doi.acm.org/10.1145/1730804.1730821. Cited on pages 1, 3, and 9.

Alexander Keller. Instant radiosity. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '97, pages 49–56, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co. ISBN 0-89791-896-7. doi: 10.1145/258734.258769. URL http://dx.doi.org/10.1145/258734.258769. Cited on page 4.

Morgan McGuire and David Luebke. Hardware-accelerated global illumination by image space photon mapping. In *Proceedings of the 2009 ACM SIGGRAPH/EuroGraphics conference on High Performance Graphics*, New York, NY, USA, August 2009. ACM. URL http://graphics.cs.williams.edu/papers/PhotonHPG09/. Cited on pages 3 and 6.

Rasterization Rules. Microsoft, 2012a. URL http://msdn.microsoft.com/en-us/library/windows/desktop/cc627092(v=vs.85).aspx. Cited on page 10.

Tessellation Overview. Microsoft, 2012b. URL http://msdn.microsoft.com/en-us/library/windows/desktop/ff476340(v=vs.85).aspx. Cited on page 10.

Jason Mitchell, Gary McTaggart, and Chris Green. Shading in valve's source engine. In *ACM SIGGRAPH 2006 Courses*, SIGGRAPH '06, pages 129–142, New York, NY, USA, 2006. ACM. ISBN 1-59593-364-6. doi: 10.1145/1185657.1185832. URL `http://doi.acm.org/10.1145/1185657.1185832`. Cited on page 8.

Martin Mittring. Finding next gen: CryEngine 2. In *ACM SIGGRAPH 2007 courses*, SIGGRAPH '07, pages 97–121, New York, NY, USA, 2007. ACM. doi: 10.1145/1281500.1281671. URL `http://doi.acm.org/10.1145/1281500.1281671`. Cited on pages 4 and 16.

Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. Stackless kd-tree traversal for high performance GPU ray tracing. *Computer Graphics Forum*, 26(3):415–424, September 2007. (Proceedings of Eurographics). Cited on page 7.

Tobias Ritschel, Thorsten Grosch, and Hans-Peter Seidel. Approximating dynamic global illumination in image space. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, I3D '09, pages 75–82, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-429-4. doi: 10.1145/1507149.1507161. URL `http://doi.acm.org/10.1145/1507149.1507161`. Cited on page 4.

Peter-pike Sloan, Jan Kautz, and John Snyder. Precomputed radiance transfer for Real-Time rendering in dynamic, Low-Frequency lighting environments. In *ACM Transactions on Graphics*, pages 527–536, 2002. Cited on page 4.

Ingo Wald and Vlastimil Havran. On building fast kd-trees for ray tracing, and on doing that in o(n log n). In *IN PROCEEDINGS OF THE 2006 IEEE SYMPOSIUM ON INTERACTIVE RAY TRACING*, pages 61–70, 2006. Cited on page 7.

Zhefeng Wu, Fukai Zhao, and Xinguo Liu. Sah kd-tree construction on gpu. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, pages 71–78, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0896-0. doi: 10.1145/2018323.2018335. URL `http://doi.acm.org/10.1145/2018323.2018335`. Cited on page 17.