UNIVERSITY OF GOTHENBURG

# Extending OGRE with Light Propagation Volumes

*Master of Science Thesis in Computer Science*

JOHAN ELVEK

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, 2012

**Extending OGRE with Light Propagation Volumes**

Johan Elvek



Examiner: Ulf Assarsson

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

Cover:
Image rendered with OGRE and the extension described in this thesis (see section 5.1 for details).

Department of Computer Science and Engineering
Göteborg, Sweden 2012

**Abstract**

OGRE is a popular open source rendering engine that offers an extensive set of core features. For more advanced rendering techniques, however, the engine must be extended in various ways.

The term *Global Illumination* (GI) is often used to describe both the effects of inter-reflecting light, and the algorithms that achieve them. Normally associated with off-line techniques, such as path tracing, recent years have seen an increase in the number of algorithms that achieve effective approximations of GI, suitable for real-time rendering.

Such algorithms are by necessity complex, and to incorporate them into OGRE is a non-trivial task. This thesis examines a number of interesting GI algorithms, and then describes the process of extending OGRE with one of them: Light Propagation Volumes.

**Sammanfattning**

OGRE är en populär och *open source* renderingsmotor som har en ansenlig mängd av grundläggande funktioner. För mer avancerade renderingstekniker, däremot, måste motorn utökas på olika sätt.

Termen *Global Illumination* (GI) används ofta för att beskriva, både effekterna av interreflekterande ljus och de algoritmer som uppnår dessa. GI associeras normalt med *off-line*-tekniker, såsom *path tracing*. Under de senaste åren har dock ett utökat antal algoritmer som uppnår effektiva approximationer av GI och är användbara inom realtidsrendering beskrivits.

Sådana algoritmer är av nödvändighet komplexa och att inkorporera dem med OGRE är en icke-triviell uppgift. Den här uppsatsen undersöker ett antal intressanta GI-algoritmer och beskriver sedan processen med att utöka OGRE med en av dem: Light Propagation Volumes.

# Contents

# 1 Introduction

When light hits the surface of an object, it interacts with that surface in a number of possible ways. The nature of this interaction depends on the characteristic properties of the given surface. While the simplest possible simulation could model surfaces as either completely reflective, or completely refractive, the different properties of real life surfaces lead to much more complex situations.

Take, for instance, the concept of subsurface scattering. When we see the face of another human being, we may think of this as our eyes receiving light directly reflected off the skin on the face. This will be true for some of the light. But, our eyes will also receive light that entered through the skin, and then bounced around a number of times before exiting again.

The effect of subsurface scattering is that we perceive human skin as soft, while a harder, smoother, material, such as steel, will have a very different visual appearance. Steel will allow very little light to refract, and the smoothness of the material will lead to fewer variations in the direction of the reflected light.

So, there is an inherent complexity in modeling light that, in part, depends on the many different properties of various materials, but that is not the end of it. In both of the examples above, we are implicitly assuming that light, from some kind of light emitter, interacts with the surface of a single object, and then travel directly to our eyes. In general, that is not the case, and the example of subsurface scattering touches upon the missing element: multiple bounces.

Most of the objects we see are not only lit by some specific light source (like a lamp, or the sun) but also from the light reflected off, or refracted through, other objects. In the context of computer graphics, the term *global illumination* (GI) has emerged as an umbrella term for describing this so called environmental lighting, as well as the algorithms that achieve the effect.

## 1.1 Off-Line vs. Real-Time Rendering

There are a number of so called off-line methods/algorithms (e.g., Metropolis light transport, photon mapping) that can be used to handle arbitrary light paths by taking the entire geometry of a scene into account during light calculations. While these techniques produce very accurate results, they are currently not efficient enough to handle dynamically changing scenes in real-time.

Modern *graphics processing units* (GPUs) on the other hand, are built to efficiently render dynamic scenes in real-time using the "z-buffer algorithm."

This algorithm handles the problem of determining visible surfaces by using a buffer of depth values, relative to the eye[1]. For each object to be drawn, the depth of the surface for each pixel is compared to the current value in the z-buffer. Surface fragments are completely discarded if determined to be behind what has previously been drawn at that pixel. Otherwise a new pixel is drawn and the current z-buffer value is overwritten.

As each surface fragment is shaded, only the geometric information of the currently shaded surface fragment is available. Even when pre-passes to render buffers containing positions, normals and other properties are performed, we cannot use that information to find geometry intersecting rays of light. At any time, only the surfaces directly visible from the eye are available.

The effect of this apparent limitation is that only *direct illumination* can be simply handled. For *indirect illumination*, more advanced utilization of current graphics hardware is required.

## 1.2 The Rendering Equation

Kajiya [1986] gives a general *rendering equation* for the intensity of light between two points, $p$ and $p'$ (the outgoing radiance from $p'$ to $p$):

$$L_o(p, p') = v(p, p') \left[ L_e(p, p') + \int_S f(p, p', p'') L_o(p', p'') \, dp'' \right]$$

$v(p, p')$ is a visibility term. If a ray from $p'$ to $p$ is intersected by some other point, $p_o$, $v(p, p') = 0$. $L_e(p, p')$ is the emitted radiance from $p'$ to $p$. The domain of integration is the set of all points $p''$ from which there is outgoing radiance, and $f(p, p', p'')$ is a measurement of the intensity of light scattered via $p'$ from $p''$ to $p$. This measurement is often calculated using a *bidirectional reflectance distribution function* (BRDF).

Since an exact calculation of the above integral is computationally hard, the task of any rendering algorithm is to provide an approximation of it for each $p'$ visible from the eye (each pixel in the final image). As previously mentioned, off-line methods take the entire scene geometry into account and are therefore lend themselves well to various approximative methods. Most importantly, they all have an inherent capability for computing $v(p, p')$ for any two given points $p$ and $p'$.

In the simplest form of GPU-based real-time rendering, and assuming a single light source $L$, however, the rendering equation is reduced to

$$L_o(E, p) = L_e(E, p) + f(E, p, L) L_o(p, L)$$

[1] Subsequent references to *the eye* should be taken to mean *the point of view of the main camera*.

for each $p$ visible from the eye, $E$, i.e., whenever we have $v(E, p) = 1$.

Fortunately, modern graphics hardware support custom shader programs and the ability to save calculated values in various types of buffers, the latter of which can subsequently be used as input by other shader programs in later passes. Through creative application of this programmability, the design of algorithms capable of providing better approximations of the rendering equation in real-time is enabled.

## 1.3 Problem Statement

Global illumination algorithms for real-time applications is a relatively new, but active field of research. In its most general application, the term GI denotes a number of various visual phenomena. But, transparency (or translucency) and shadows cast by directly illuminated objects — while arguably constituting GI phenomena — are notably such important parts of graphics programming, that methods for handling these have long been considered their own subjects.

The visual appearance of scenes rendered with direct illumination only, is that of a number of light-receiving surfaces that reflect some light toward the eye while completely absorbing the rest. To alleviate this situation, the traditional strategy is to add a constant ambient term to the diffuse and specular ones, even if they are zero. The result is that non-illuminated surfaces are uniformly brightened, regardless of the scene geometry and position of light sources.

Generally speaking, we can identify two categories of indirect illumination: low-frequency illumination (diffuse inter-reflections), and high-frequency illumination (caustics, glossy reflections). Due to the need for more information, high-frequency phenomena are also more difficult to approximate.

Most of the methods in section 2 focus on, or are exclusively limited to, diffuse indirect illumination. Furthermore, Tabellion and Lamorlette [2004] have shown that single-bounce diffuse inter-reflections are often enough to provide convincing global illumination, even for something as visually demanding as an animated feature film.

### 1.3.1 Goal of the Thesis

This thesis work is done in association with EON Reality, Inc., which offer 3D visualization and Virtual Reality solutions for businesses and educational institutions. The next major version of their rendering engine will be based on heavily extending the core features of OGRE [Torus Knot Software Ltd,

2011], an open source rendering engine that supports multiple graphics APIs, notably OpenGL and Direct3D.

The goal of the thesis is to identify a state-of-the-art method that offers a good balance between efficiency and accuracy, and subsequently extend OGRE with this algorithm. Since a large number of models and scenes for the current EON engine already exist, an important consideration is to find a technique that requires little, to no, pre-processing.

# 2 Related Work

In the following subsections, extensive summaries will be given for some of the more interesting techniques in the relatively short history of algorithms for approximating indirect illumination in real-time rendering.

## 2.1 Shadows

As noted previously, the shadows resulting from direct illumination is generally not considered to be part of the problem of global illumination. But, since a number of GI algorithms are built on a particular shadow method, this section will briefly cover the basics.

For real-time rendering, there are two common techniques for adding shadows to a scene. Crow [1977] suggests using silhouette edges of scene geometry to construct *shadow polygons*, subsequently used to determine what parts of the geometry that are shadowed. The other common method, however, is the one of particular interest for later parts of this thesis, and that is shadow mapping [Williams, 1978].

Shadow mapping is done by first rendering the scene from the point of view of the light, writing depth values (normally in clip space) into a *shadow map*. When rendering the scene from the point of view of the eye, the shadow map coordinates of each surface fragment, as well as its depth in the appropriate space, are calculated. The actual depth stored in the map is then compared with the fragment's value to decide whether the fragment is in shadow or not.

Although shadow mapping is not physically accurate, variants of this technique are capable of achieving feasible results in many situations. It also effectively utilizes some of the most important capabilities of the GPU.

## 2.2 Pre-Computed Lighting

While ultimately not appropriate for this thesis, many interesting GI approaches involve various pre-processing steps. The result of the pre-processing is used during execution of the real-time application to simulate, for instance, diffuse inter-reflections.

Among the simplest methods are environment maps, which are queried during shading for light coming from a specific direction, and light maps, which are textures describing surface irradiance [Akenine-Möller et al., 2008]. These methods are simple in terms of usage during real-time application, but the pre-processing involved may be arbitrarily complex and produce extremely accurate lighting. Important constraints are that all occlusion must

be precomputed, and that the reflections do not affect dynamic objects.

A very influential paper is *The Irradiance Volume* by Greger et al. [1998]. This method takes advantage of the fact that irradiance as a function of both direction and position is largely continuous[2].

Approximative irradiance values are stored in a grid, adaptively subdivided to make a finer grid wherever geometry is present. The values at the closest grid vertices for a given surface fragment are interpolated during the lighting calculations. Irradiance volumes have an advantage over light maps, in that dynamic objects are affected by the indirect illumination from static geometry. Naturally, the reverse is not true.

In another seminal paper on *Precomputed Radiance Transfer* (PRT), Sloan et al. [2002] use *spherical harmonics* (SH) to store *transfer functions* in a number of samples over an object's surface. Assuming a low-frequency lighting environment, the transfer functions map from (any) incoming radiance to the correct, outgoing, radiance. This mapping can take both self-occlusion and local inter-reflections into account. The basic concept can be extended to *some* possibility for inter-reflections between objects as well, but works best for a single, designated "sender" object and one receiver object.

As the mathematical details of spherical harmonics are beyond the scope of this thesis, the interested reader is referred to [Ramamoorthi and Hanrahan, 2001] or [Sloan, 2008] (the former of which also presents an efficient scheme for creating irradiance environment maps). An introduction to the subject is also available in [Akenine-Möller et al., 2008]. The basic concepts will be expanded upon in section 3.2.

## 2.3 Ambient Occlusion

Using a constant ambient term for background lighting results in an absence of details for surfaces not directly illuminated. In real life, though, things like corners in a room, cloth folds, and carved details on a piece of furniture are often perfectly distinct, even if only inter-reflected light is reaching those surfaces. The reason for this is that for a given point on a surface, the incoming background light may be partially occluded by other parts of the surrounding geometry.

In computer graphics, the effect of this occlusion is known as ambient occlusion (AO). One possible strategy is to calculate the self-occlusion for each object in a modeling program, and bake the resulting values into a texture. This procedure would result in a very accurate and detailed AO,

[2]Spatial discontinuities exist on the boundaries between non-shadowed areas and penumbras, as well as between the penumbras and umbras.

but the contact shadows between distinct, but closely situated, objects would be lost.

An early method for fully dynamic AO was presented by Bunnell [2005]. During occlusion computation, each object is seen as a set of surface elements. A surface element is stored per vertex, represented by position, normal, and area, as an oriented disk. The area of each element is defined as the sum of all $A_t/3$, where each $A_t$ is the area of a triangle $t$, sharing the vertex in question.

First, surface elements receive shadows from other elements by calculation of an *accessibility value*. This value may be used for final occlusion, but leads to exaggerated AO, as surface elements that are heavily occluded will affect surrounding elements too much. Instead, a second pass can be performed, where pairwise form factors between elements are calculated, and then modulated by the accessibility value of the first pass.

The method can also be extended to determine the *bent normals* ("average incoming light vector," [Landis, 2002]), which enable directional occlusion, and simple near-field color bleeding if the color of each surface element is available. Bunnell's method results in very accurate AO, but needs to be furnished with a hierarchical level-of-detail (LOD) scheme to be feasible even for moderately complex scenes. Obtaining the area of each surface element also demands some light pre-processing.

A much simpler, but also faster, technique was presented in [Mittring, 2007], and more extensively in [Kajalin, 2009]. Dubbed *Screen-Space Ambient Occlusion* (SSAO), the method differs significantly from the previous method in that it does not use any other geometrical information than that present in the final image (or screen) space.

For each pixel, its depth value is obtained from the z-buffer and compared to the depth values of a small number (usually 8 or 16) of sampled points. If a sampled point is closer to the eye than the current pixel, it is regarded as an occluder. A simple depth range check is used to avoid nearby objects shadowing objects on the horizon.

The small number of samples introduce banding in the final result, so each (pre-computed) offset vector is rotated using a random 4x4 pattern of rotation vectors. The rotation trades banding for noise, and the same, distinctly recognizable, noise pattern is repeated every 4x4 pixels.

The advantage of using such a small random pattern, however, is that the resulting noise can be blurred away by using an equally small blur kernel. Rather than using a simple box-filter, a cross-bilateral and edge-aware blur that blends only pixels of similar screen space depth is used.

As SSAO is a screen-space technique, it suffers from problems such as previously occluded surfaces becoming brighter as their occluders move out-

side the camera view, and vice versa. Furthermore, the simple scheme for calculating occlusion, and its way of dealing with the resulting self-occlusion, leads to concavities becoming dark, edges becoming bright, and unoccluded surfaces becoming gray. The resulting AO is not physically accurate, but can nonetheless be seen as a particular artistic appearance.

A similar, but somewhat more sophisticated, screen-space technique called *Screen-Space Directional Occlusion* (SSDO) is given by Grosch and Ritschel [2010]. Instead of using only depth, both position and normal are read from a pre-computed screen buffer. This method also lets the user set a value for maximum radius of influence, allowing for more detailed occlusion by nearby surfaces, or more diffuse occlusion by relatively far away objects.

All offset vectors describe points in the hemisphere above the surface, and the points sampled through the 2D projection of those are distinguished by whether the offset vector described a point below, or above, the surface sample. Only the samples where the offsets were below the sampled surface are counted as occluders, and thus both self-occlusion and brightened edges are avoided, leading to a more realistic approximation of AO.

Given some representation of directional background lighting (e.g., an environment map), SSDO can also handle directional occlusion. In a deferred shading system (see section 4.1) the color buffer can be used to let occluders bleed color onto the occluded surface. This is a very rough approximation (as the color bleeding surfaces may actually be totally occluded) for near-field indirect illumination.

In [McGuire, 2010] a geometry shader is used to construct *ambient occlusion volumes* (AOV) on the fly for each rendered primitive/triangle. As each such volume $B$ is rasterized, values from pre-computed geometry buffers at the rasterized pixel are read, which gives the position and normal of a visible surface point $x$. Using the available information on $B$ and $x$, an accessibility value is decreased according to the value of a falloff function $g$.

An occlusion volume can be thought of as a pie slice with the base on the same plane as the triangle, extending in all other directions by a maximum distance of influence $\delta$. This enables occluding geometry that is not visible on the screen to affect the visible objects, as the occlusion volume may still be rasterized. A possible optimization is to construct the volumes for static scenery once (which would also allow quads rather than triangles), and simply render these during AO computation.

The accessibility value for each pixel is initialized to 1 and decreased by the value of $g$ each time rasterization of an occlusion volume affects that pixel. The accessibility value is saturated at 0, which means total occlusion. Thin objects ($< \delta$) in close proximity ($< \delta$) may artificially accelerate the decrease and lead to over-occlusion. The final occlusion values can be remapped, either

according to a pre-computed curve, or according to an arbitrary parabolic curve, but fully (and possibly falsely) saturated values will remain so.

While not as fast as the various image-space methods, AOVs produce results nearly on par with ray traced AO in many cases. As is often done for image-space AO, the occlusion volumes can also be rendered at a lower resolution and then upsampled at the cost of some quality.

## 2.4 Dynamic Pre-Computation and Off-Line Inspired Methods

While true off-line methods have begun to take advantage of the computing power of graphics hardware (for instance, van Antwerpen [2011] efficiently deals with stochastic termination when performing random walks for Monte Carlo-based rendering on GPUs), there have also been successful attempts of applying the concepts of either off-line, or pre-computed, techniques to real-time rendering.

In [Nijasure et al., 2005], the entire scene is divided into a three dimensional grid and then rendered into small cube maps at each grid point. The cube maps represent the *radiance field* at the corresponding grid points.

Taking each pixel of a cube map into account, *spherical harmonics coefficients* are subsequently calculated and stored in a volume texture. The indirect illumination for each surface fragment is computed by evaluating a tri-linear interpolation of the SH coefficients, and the result can also be used in rendering the cube maps anew in order to take multiple bounces into account.

Inspired by photon mapping, McGuire and Luebke [2009] renders *bounce maps*, which represent the first bounce of light, from the point of view of each light source. These are used as input for a CPU-bound photon trace, while the hardware concurrently uses deferred shading to render direct lighting.

Final gathering is then performed in screen space, by converting the result of the photon trace into *photon volumes*, which are *splatted* onto all visible surfaces. The bounce maps and photon volumes are GPU-friendly alternatives to the otherwise expensive initial bounce and final gathering steps of the photon mapping algorithm.

## 2.5 Instant Radiosity and Reflective Shadow Maps

Originally suggested by Keller [1997], the Instant Radiosity algorithm treats a subset of all light-reflecting surface points as *virtual point lights* (VPLs) ([Laine et al., 2007], [Ritschel et al., 2008]). In the original algorithm, the

scene is rendered, using shadow volumes, multiple times from random points on the surface of an area light. Through a CPU-bound quasi-random walk of the scene based on Monte Carlo integration (*or* simple fixed length paths), a number of VPLs are identified. The scene is again fully rendered, with shadows, from the viewpoint of those, and the final frame is acquired via a composite of all rendered images.

Laine et al. [2007] limit themselves to single-bounce indirect lighting, and use 180° spot lights, or point lights, rather than area lights. VPLs are created via a sampling scheme based on so called Delaunay triangulation, resulting in a Voronoi diagram where a point in each cell marks the origin for a ray towards the VPL. For each VPL, a parabolic shadow map is rendered[3].

All VPLs[4] are created at the beginning of the application run, and again whenever old ones are invalidated. Invalidation occurs when the location of a VPL ends up outside the lit region (spot lights only) or a VPL becomes occluded. By *only treating static geometry* as occluders for the indirect occlusion, invalidation can only occur as a result of a moving light source. In other words, dynamic objects may receive indirect lighting but do not contribute to it.

Each VPL acquires the color of the surface point at which it originates, sampled from a heavily blurred texture to avoid small texture details having too great an effect on the indirect illumination. The power of a VPL is proportionate to the size of its Voronoi cell, and the sum of all powers equals that of the original light source.

To effectively deal with shading for such a large number of lights, an interleaved sampling scheme is employed. In a deferred shading step, the whole frame buffer is divided into a large number of small tiles and a subset of all virtual point lights are assigned to each cell. This results in a noise pattern that is removed in a filtering step when assembling the final image.

Ritschel et al. [2008] note that the nature of indirect illumination allows for the visibility term to be roughly approximated, and introduce *Imperfect Shadow Maps* (ISM). The method requires pre-processing the geometry by creating a simple point representation of each object. Many (e.g., 1024) VPLs are created, simply by sampling the shadow map of the original light source, and small parabolic shadow maps are then created for each by splatting the point representations into the depth buffer.

The above scheme allows for a large number of ISMs to be quickly rendered, but leaves holes in the shadow maps. This is alleviated by performing

[3]Parabolic shadow maps capture the entire hemisphere from the viewpoint of the camera.

[4]256 is a suggested number.

a *pull-push* operation, where the shadow maps are first downsampled and the finer levels then filled in by interpolating the values in the downsampled maps.

Shading of the final scene is performed by interleaved sampling, as for the previous method. ISMs also support glossy reflections and area lights. Improvements on the original algorithm was presented in [Ritschel et al., 2011], which introduces a view-adaptive sampling scheme for creating the VPLs and varying the density of the point representations according to distance. Both strategies make sure that each VPL is better utilized, leading to higher quality results.

The observation that pixels in a shadow map are exactly those surface fragments that cause indirect illumination is taken advantage of in [Dachsbacher and Stamminger, 2005]. Instead of creating VPLs from the shadow maps, positions, normals and flux are stored along with depth in a *Reflective Shadow Map* (RSM).

During shading, each visible point $p$ is projected into the shadow map, and indirect illumination is gathered from nearby pixels. The normal of $p$ is compared to each sample to make sure that indirect illumination only arrives from the hemisphere above $p$. Note, however, that this gathering step completely ignores the visibility term.

To get consistent, non-flickering, illumination, a large number of samples have to be taken and the flux stored in the RSM must be calculated using blurred textures. The number of necessary samples is still large enough that a pre-pass on a low resolution image is performed, and the result of that later interpolated. The interpolation scheme works such that the information of the low resolution pass may be deemed insufficient for a particular point. In that case, a full gathering step is performed.

An alternative to the gathering approach was given in [Dachsbacher and Stamminger, 2006]. By using an importance sampling scheme, a set of pixel lights is drawn from the RSM, and each element is then splatted onto the screen buffer using volumes shaped according to the nature of the pixel light. Splatting works similar to AOVs, in that the rasterized volume describes the area of influence (for the indirect illumination, in this case).

For mainly diffuse surfaces, the splatted volume will have an egg-shape, but the technique also adds some support for high-frequency phenomenon such as caustics (by narrowing the shape of the volume where appropriate). An available AO buffer can also be used to reduce the number of needed splats.

To increase the speed of the splatting process, Nichols and Wyman [2009] proposed to subdivide the camera view image based on discontinuities in depth and normal values. Regions where those values change slowly may be

splatted using low resolution images which are later upsampled. For either splatting method, the visibility term is disregarded.

## 2.6 Volume-Based Methods

Under the heading *Light Propagation Volumes in CryEngine 3* in [Tatarchuk et al., 2009], Anton Kaplanyan suggested an alternative use for RSMs by projecting the stored surface normals into SH coefficients, which are then injected into *Light Propagation Volumes* (LPVs). Further development of the idea was later presented by Kaplanyan and Dachsbacher [2010], and again in [Kaplanyan et al., 2011].

LPVs are represented as volume textures where each volume element, or voxel[5], in turn represents a cubic region of space in the scene. After the light has been injected, it is iteratively propagated through the volume. The propagation scheme consists of having the light in each voxel, or cell, propagate onto the faces of the cells lying in the 6 main axes of the 3D space.

To account for occlusion of the inter-reflected light, a *blocking potential* for geometry rendered into an RSM can be calculated and injected into a second *geometry volume* (GV). During propagation, the blocking potential is taken into account when calculating how much light is propagated to the neighboring cells.

As propagated light can quickly move away from the geometry stored in the RSM, the geometry visible from the eye may also be injected into the GV. Doing so, however, also makes the visibility term partially screen-space dependent. This, in turn, leads to instances of flickering as moving the camera may result in previously blocking geometry disappearing from the main view.

The result of each iteration is accumulated, and the final result is used to shade the scene. Because of the low-order SH used, in combination with the highly discretized light transport, the technique is inherently only suited for low-frequency diffuse inter-reflections.

A similar approach was given by Papaioannou [2011], but instead of propagating the light, the injected SH coefficients represent *radiance field evaluation points* denoted *Radiance Hints* (RHs). The first bounce is unoccluded, although a heuristic for visibility computation was suggested as an optional extension. Multiple depth values, $d_i$, along a ray between the reflecting surface fragment and the associated RH are sampled in screen-space. Each sample is compared to the depth, $d_s$, in a geometry buffer, and if $d_s > d_i$, the inter-reflected light is attenuated by a constant factor.

[5]This is the 3D analogue of a 2D texture element, or texel.

Multiple bounces are encoded by performing subsequent passes where RHs are updated through the sampling of other RHs. Here, a stochastic method for approximating the visibility term is used, based on distances between the RHs, and the maximum and minimum distances to injected RSM samples for each. Shading consists of interpolating the contributions of nearby RHs.

A very different approach to volume-based GI was presented by Thiedemann et al. [2011]. It builds upon the idea of *scene voxelization*, where a discrete representation of the scene geometry is created and then stored in a volume texture. Static geometry may be voxelized once, and a copy of the result be used to initialize the scene representation each frame.

Apart from presenting a number of variations for global illumination, the voxelization method described is novel itself. Given a mapping between objects and coordinates for a texture atlas, the voxelization process compares favorably with similar techniques in terms of both efficiency and quality. The world space positions of each object are stored in their respective texture atlases using depth peeling. This information is then used to mark the regions in the scene, as represented by the volume texture, that are occupied.

A suggested ray/voxel intersection allows for a number of different methods of indirect illumination. For a near-field single bounce illumination, a number of rays is cast from the shaded surface, and upon intersection, the reflected light is read from an RSM. Storing additional information (normals and BRDFs) in the volume texture enable a voxel-based path tracer. The complexity of the intersection test depends on the length of the ray, however.

# 3 Light Propagation Volumes

This section contains a more in-depth description of light propagation volumes. Figure 1 visualizes the propagation scheme of the algorithm. The images are rendered using the LPV extension of OGRE described in section 4.

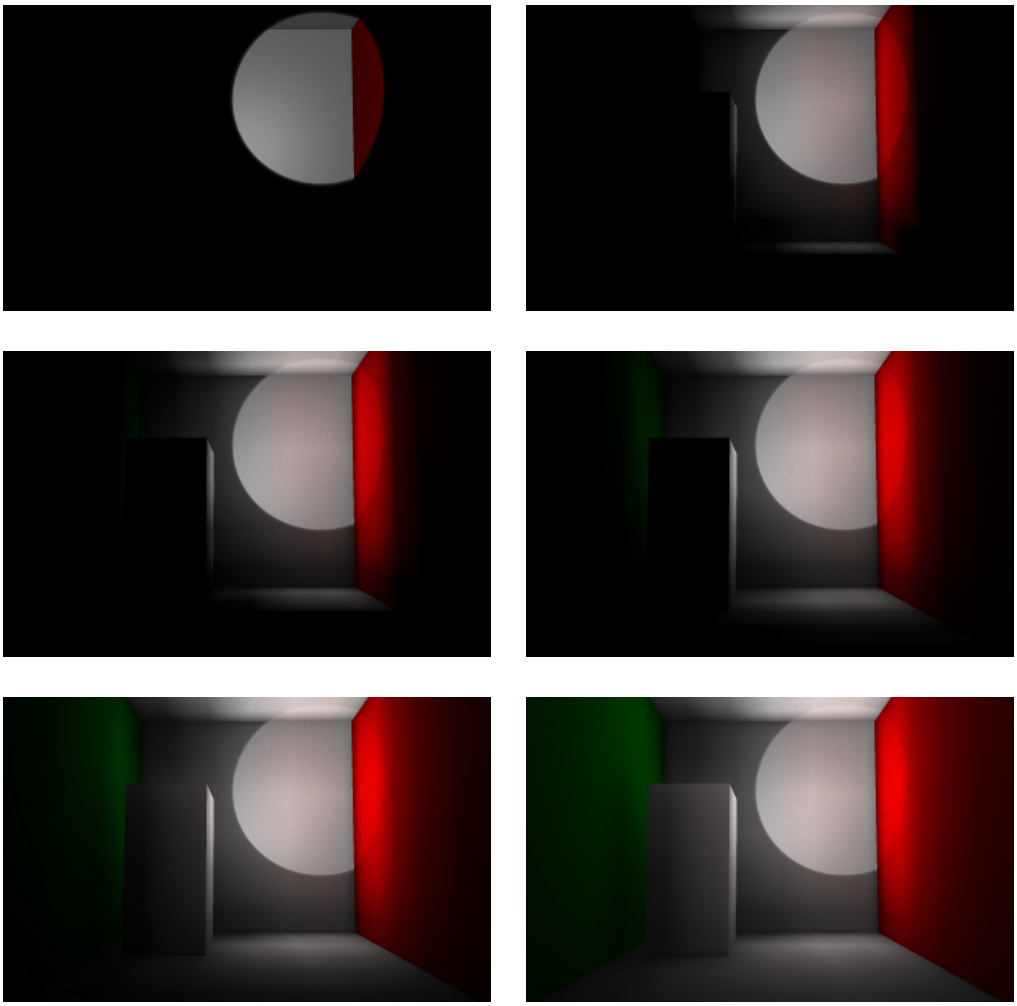

Figure 1: Cornell box rendered using the implementation described in section 4. From left to right, top to bottom, the number of propagation iterations are: 0, 1, 2, 4, 8, 16.

Although a number of the algorithms described in the previous section offer a more accurate approximation to the rendering equation, in some cases capable of various high-frequency phenomena, light propagation vol-

umes stood out as the most interesting. This was mainly due to two reasons; speed, and lack of pre-processing steps.

LPVs are part of game developer Crytek's CryENGINE 3, which is a multi-platform rendering engine supporting DirectX, Sony PlayStation 3, and Xbox 360. Since gaming consoles have strict constraints on performance, any GI solution supported by such must be both efficient and stable.

## 3.1 Reflective Shadow Maps

RSMs are created during the standard shadow mapping phase, but uses a *multiple render target* (MRT) instead of a standard render target. MRTs have several buffers, or surfaces, which can be rendered into. In addition to any regular shadow map data, the RSMs used with LPVs store position, normal, and reflected flux.

Reflected flux is calculated by multiplying the diffuse color with the clamped dot product of the normal and direction to the light source. The various available sources provide conflicting information on whether to weight this value further, or not.

For the implementation described in section 4, either way is fine. However, weighting reflected flux by the texel area in world space (c.f. geometry injection, below) seems to give better results for cases where there are large depth differences in the RSM.

The original RSM technique, described in section 2.5, uses blurred versions of diffuse textures, but LPVs are less sensitive to details in the texture maps. Partly because all RSM texels will contribute to the indirect illumination, and partly because the propagation itself will blur the reflected light.

To make sure that the injection step is fast, RSMs need to be of low resolution (possibly downsampled). Injection consists of rendering a point, or vertex, cloud consisting of as many elements as the RSM has texels. For each vertex, an RSM surface sample (surfel) is obtained, and injected into the LPV.

## 3.2 Spherical Harmonics

Spherical harmonics (SH) have an extensive number of applications in various fields, e.g., physics (gravitational and electric fields) and computer graphics (BRDF representations) [Sloan, 2008]. As spherical harmonics are a fairly complex topic, the following is a practical approach to describing the basics, as they apply to the context of light propagation volumes.

SH are orthonormal basis functions, which can be used to represent "scalar functions on the unit sphere" [Akenine-Möller et al., 2008]. Informally, we can think of what this means in the following way: Given a unit direction vector, and a SH representation of some function, we can use this information to obtain the value of that function in the given direction.

A SH representation is a vector of coefficients, most often denoted as $Y_{lm}$ (or $Y_l^m$) where $l \geq 0$ and $-l \leq m \leq l$ [Ramamoorthi and Hanrahan, 2001]. The subscript $l$ gives the number of *bands* of the SH representation, which in turn describe the degree of polynomials by which the function is approximated ($l = 1$ is a linear polynomial, $l = 2$ a quadratic, and so on).

The smaller the number of bands, the more approximate the representation is. For relatively small values of $l$ we say that we have *low-order* SH. LPVs use $l = 2$, and are thus low-order (for comparison, PRT use $l = 3$ for diffuse lighting, and $l = 5$ for glossy reflections, which are also low-order SH).

The number of coefficients needed is given by $l^2$, so in the case of LPVs that means that the SH coefficients (*for each color*) can be stored in a texture with 4 components (RGBA). Rather than using the $Y_{lm}$-notation, we will write $\mathbf{c}$ to denote a SH coefficients vector with 4 components, $(c_0, c_1, c_2, c_3)$.

Given a SH coefficients vector, and a direction vector, evaluation is as simple as a dot product of the coefficients vector and the SH *projection* of the direction. In the context of LPVs, if we have the SH coefficients $\mathbf{c}_R$, representing the directional distribution of red light at some point, and a direction vector, $v = (x, y, z)$, the intensity of red light in direction $v$ is found by dotting $\mathbf{c}_R$ with the SH projection of $v$. The SH projection, $\mathbf{c_v}$, of $v$ is,

$$c_0 = \frac{1}{2\sqrt{\pi}}$$

$$c_1 = -\frac{\sqrt{3}}{2\sqrt{\pi}}y$$

$$c_2 = \frac{\sqrt{3}}{2\sqrt{\pi}}z$$

$$c_3 = -\frac{\sqrt{3}}{2\sqrt{\pi}}x.$$

## 3.3 Algorithm

Given a surfel, its position is used to find the correct cell of the LPV. As injected surfels will effectively be stored in the center of its cell, each surfel is offset half a cell size in the direction of its normal, to avoid potential self-occlusion. For each color component (i.e., red, green blue), SH coefficients

for a clamped cosine lobe centered about the sample normal, multiplied by the correspondent flux, represents the directional distribution of said color.

The four SH coefficients, $(c_0, c_1, c_2, c_3)$, for the clamped cosine lobe are easily acquired by a function that maps the $xyz$-components of a surfel's normal in the following way:

$$c_0 = \frac{\sqrt{\pi}}{2}$$

$$c_1 = -\sqrt{\frac{\pi}{3}}y$$

$$c_2 = \sqrt{\frac{\pi}{3}}z$$

$$c_3 = -\sqrt{\frac{\pi}{3}}x$$

Once injection is complete, an iterative propagation scheme is performed. Conceptually, each cell propagates light to all 6 neighbors along the positive and negative main axes. In terms of shader programming, however, each voxel $v$ gathers the light from its neighbors.

### 3.3.1 Propagation and Lighting

For each face of $v$, the amount of incoming flux, $\Phi_f$, is computed. The flux is propagated, from the center of the neighbor, in direction $w_f$, towards the center of the face. The intensity, $I(w_f)$, is evaluated by taking the dot product of the neighbor's sampled SH coefficients and the SH projection of $w_f$. We then compute the flux as $\Phi_f = \Delta w/4\pi \cdot I(w_f)$, where $\Delta w$ is the subtended solid angle to the face. Figure 2 is a schematic image, in 2 dimensions, of the propagation.

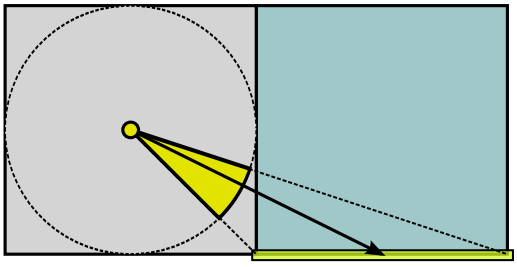

Figure 2: Schematic image of a neighbor cell, on the left, propagating light onto the bottom face of another cell.

The incoming flux at each face is *re-projected* into a directional distribution at the cell center. This is done by assuming a point light in the center

of the $v$, with flux $\Phi_l = \Phi_f/\pi$, directed towards the face. A clamped cosine lobe about the direction vector is projected into SH coefficients, which are multiplied with $\Phi_l$.

The above is done for each color component, for all faces[6], and neighbors. All SH coefficients are added and written to the outputs. The result of each iteration is used as input for the next iteration, but also added to an accumulation buffer. When propagation is finished, the accumulation buffer represents the diffuse indirect illumination.

During shading, the indirect illumination is evaluated by projecting the shaded surface fragment's negative normal into SH coefficients, and taking the dot product of those with the SH coefficients sampled from the accumulation buffer.

In some cases, artifacts such as light bleeding through directly lit geometry may occur. For a shaded surface point $x$, this can be alleviated to some extent by dampening all sampled SH coefficients, $\mathbf{c}(x)$. The directional derivative, $\nabla_n \mathbf{c}(x)$, in the direction of the surface normal, $n$, can be calculated in the following fashion:

$$\nabla_n \mathbf{c}(x) = \mathbf{c}(x + \frac{n}{2}) - \mathbf{c}(x - \frac{n}{2})$$

Whenever $\mathbf{c}$ and $\nabla_n \mathbf{c}(x)$ are deviating (their dot product is less than zero), $\mathbf{c}(x)$ should be dampened in some appropriate fashion.

## 3.4 Fuzzy Occlusion

To make sure that light is less likely to bleed through objects, a scheme denoted *fuzzy occlusion* is used to approximate the visibility term. The same SH coefficients that are scaled by the flux and injected into the LPV, can instead be scaled by a blocking potential, $B(\omega)$, and injected into a geometry volume (GV). Given, $\omega$, the direction to the light source[7], and a surfel, $s$, the blocking potential is calculated as,

$$B(\omega) = \frac{A_s \langle n_s | \omega \rangle}{s_2}.$$

$A_s$ is the area of the surfel, and can be calculated as the size of the texel in world space. $\langle n_s | \omega \rangle$ is the cosine of the angle between the surfel's normal and $\omega$. $s_2$ is the world space size of a cell in the volume.

[6] Except the face that borders on the neighboring cell.

[7] Or the camera, rather, but this is the same vector for RSMs.

Corners of the GV are centers of the LPV, and vice versa. When light is propagated through the face that borders neighboring cells, the GV coefficients of the four corners of that face are linearly interpolated as $\mathbf{c}_{GV}$. The direction from the neighbor to the face is projected into SH coefficients, $\mathbf{c}_d$, and incoming flux is then scaled by 1 minus the dot product of $\mathbf{c}_{GV}$ and $\mathbf{c}_d$.

Since the propagated light will quickly move beyond the view of the light source, occluding geometry may not be represented in the RSMs. Therefore, main camera geometry is also injected. To make sure that occluders are not added twice, separate GVs need to be maintained. The GVs are later "combined" into a single GV by simply taking the maximum of the coefficients. Adding camera geometry provides a much better approximation of the visibility term, but flickering may occur as geometry appears in, and disappears from, the main view.

## 3.5 Cascaded LPVs

A very important extension, primarily for use with directional lights, is the use of several nested, or cascaded, LPVs. The reasoning behind this extension is similar to that of *cascaded shadow maps* (see section 4.2.1). In short, cascaded LPVs are used for directional lights (since they potentially affect large portions of the visible scene) to provide detailed indirect illumination close to the eye, while areas further away use larger volumes for a more approximate result.

The cascaded LPVs move with the camera, to make sure that the nearby surfaces always receive the more detailed lighting. As potential flickering issues, correct blending and creation of geometry volumes all require careful attention, the task of adding cascaded LPVs may become quite involved. Due to time constraints, this very useful addition was unfortunately not included as part of the implementation described in section 4. The interested reader is referred to any of the sources on LPVs mentioned in section 2.6, especially the two most recent ones.

# 4 Implementation

This section describes the process of extending OGRE with light propagation volumes. Although OGRE offers a lot of built-in functionality in terms of rendering engine basics, it is geared towards traditional forward rendering (see below). This had the effect that some extensive work had to be made in order to build a foundation, upon which to base the LPV implementation.

Otherwise, OGRE is an extensive rendering engine that, among other things, provide a scene manager, mathematical libraries, and a custom model and animation format. The material system includes support for scripting materials, as well as giving specific shader parameter values and preprocessing directives.

Various graphics APIs (notably OpenGL and Direct3D 9) are supported through plug-ins called *render systems*. In addition to supporting GLSL and HLSL, NVIDIA's Cg is also supported (the latter of which was used to write the shaders for this implementation). An effort was made to keep the implementation compatible with both OpenGL and Direct3D 9. While this ultimately failed in the case of LPVs (see section 5.1), everything else has been tested for both render systems.

## 4.1 Deferred Shading

The "traditional" z-buffer-based rendering is often referred to as forward rendering. It may consist of either shading each object with a shader that accesses many light parameters, or iterating through the lights and shading each affected object. If there are a large number of lights, forward rendering can quickly become expensive, as objects are shaded and then overwritten by subsequent objects. The first approach may also lead to rather large, complex, shaders.

Deferred shading (see for instance [Shishkovtsov, 2005] and [Koonce, 2007]) represents a different approach. First, geometric information is rendered into a number of buffers collectively called the geometry buffer (or G-buffer). Typically, a G-buffer consists of positions, normals, *albedo* (texture mapped color value), and possibly other material-related information. After creation, the G-buffer is used as input during light shading, to make sure that only the visible geometry is shaded for each light.

Also important is that many of the methods described in section 2 either benefit greatly from, or are entirely dependent on, the information normally stored in a G-buffer (SSAO, splatting RSMs, LPVs, etc.). OGRE is most naturally used as a forward renderer, however. There exists a deferred shading demo, which works rather well, but is incomplete (e.g., directional lights

cast no shadows) and suffers somewhat from an unnecessarily complicated design.

Instead of modifying the demo, a simpler approach is chosen. Adding objects bound for the G-buffer to a specific *render queue*, a relevant *material scheme* name to the material scripts as well as the G-buffer render target, and writing shaders, is enough to get a basic deferred shader working.

By encoding view space position as normalized depth, [Pettineo, 2010], and the view space normal as its $xy$-coordinates, [Pranckevičius, 2009], both vectors can be stored in a single texture buffer with three 16-bit floating point components. A second buffer is used for the albedo. An example of G-buffer contents can be seen in figure 3.

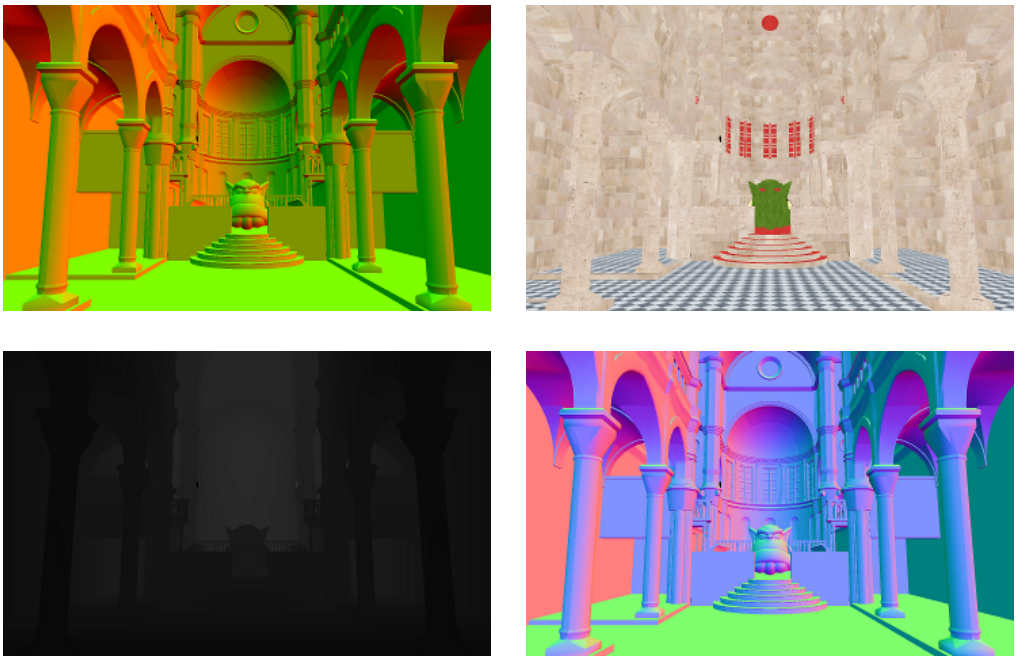

Figure 3: Top left corner shows the encoded position and normal. Top right corner shows the albedo. In the bottom row, from left to right, we have the (encoded and) normalized view space depth, and the decoded view space normal

### 4.1.1 Lights

A full-screen quad can be rendered for each light, as the rasterization of such an object leads to each pixel in the image being shaded. Range and attenuation can be handled by the shader to make sure that only the appropriate surface fragments are affected by the light.

A sounder strategy, however, is to send simple volumes, shaped like each light source's region of influence, down the pipeline. This will (roughly) ensure that only the parts of the scene that each light affects is rasterized. For spot lights we can use cones, for point lights (not used in this implementation) we use spheres, and the full-screen quad is reserved for the directional lights only (as those are generally used for simulating sunlight).

Further optimization is done by alternating between culling the back- and front-faces of the volume, depending on whether the eye is outside or inside it — the z-buffer comparison is adjusted accordingly. Now volumes may be occluded by other geometry when the eye is on the outside, and no area outside the volume is rasterized when the eye is on the inside.

Adding this setup is similar to setting up the deferred shading (render queue, material scheme), but each light volume must also be associated with a specific set of light parameters, as well as a specific shadow map. The former can be handled by registering as a listener for render system light queries. This allows for a singleton list (containing the appropriate light object) to be returned as the list of lights affecting the object.

Unfortunately, a problem arises with OGRE's built-in shadow mapping, as the order of the shadow maps is decided elsewhere. The custom light list must still respect this order, up to the number of shadow casting lights, if using the built-in shadows. The deferred shading demo solves this by iterating over the lights, pause rendering, prepare the shadow map, and resume rendering, making only a single shadow map available at any time.

Rather than mimic that solution here, two reasons for replacing the built-in shadow mapping entirely can be identified:

- the built-in system *only allows for a single buffer in the shadow map render target* — an RSM needs a *multiple render target* with many surfaces to write to

- a custom system seems to facilitate a more fine-grained control of *cascaded shadow mapping* (see below)

As OGRE does offer the possibility to override basic shadow camera setups, the second reason is perhaps of minor significance. Still, overriding the camera setup means handling several shadow maps for the same light, in addition to handling the light itself (and the associated light volume) in the manner above.

The adopted solution, instead, is to simply wrap light, volume, and camera in a single class. This gives considerable control without the need to grasp the finer details of the built-in shadow system (possibly through examining the source code).

## 4.2 Shadow Mapping

Regular shadow mapping, based on straight-forward depth comparisons, can suffer from various artifacts, such as shadow acne[8] and aliasing/jagged edges of shadow borders. The former must be dealt with by careful biasing of the compared values, and the latter by excessive sampling of the shadow map.

Another, easily implemented, shadow map technique is the *Variance Shadow Map* (VSM) [Lauritzen, 2007]. Rather than storing single depth values for later comparison, the view space depth, $x$, and its square are stored as the first and second moments, $E(x)$ and $E(x^2)$, of some local depth distribution.

During shading of a surface fragment with depth $t$, its shadow map coordinates are found and the moments at that point are sampled. Now, the mean, $\mu = E(x)$, and variance, $\sigma^2 = E(x^2) - E(x)^2$, can be used to calculate the upper bound of $P(x \geq t)$. If $t \leq \mu$, the upper bound is 1, and we assume that the surface fragment is fully lit. Otherwise, Chebyshev's inequality is used to find the bound,

$$p_{max}(t) = \frac{\sigma^2}{\sigma^2 + (t - \mu)^2}.$$

The value of the upper bound can simply be used, as a visibility term, to modulate the outgoing radiance.

A major benefit of the distribution representation over depth values for exact comparison, is that the shadow maps can be linearly filtered (e.g. blurring, anisotropic, or view-dependent, filtering). One caveat, however, is that of overlapping occluders. In essence, the lighter penumbras of occluders closer to the light may leak through occluders further away. The situation can be somewhat mitigated by setting all values of $p_{max}(t)$ below some threshold $\tau$ to 0, and remap all other values to be in the interval [0..1]. Large values of $\tau$ lead to "thickening" of the shadows. Examples of both filtering and light leaking can be seen in figure 4.

### 4.2.1 Cascaded Shadow Maps

As directional lights usually cover large outdoor spaces, using a single shadow map is often insufficient in terms of both quality and efficiency. Unless the combination of a particular scene and camera movement constraints allows for a single, static, shadow camera, cascaded shadow maps (CSMs) are a useful alternative for outdoor lighting.

[8] A commonly used term for denoting the tendency of precision problems to give rise to incorrect self-shadowing. This can present itself as, e.g., black stripes over flat, directly lit, surfaces.

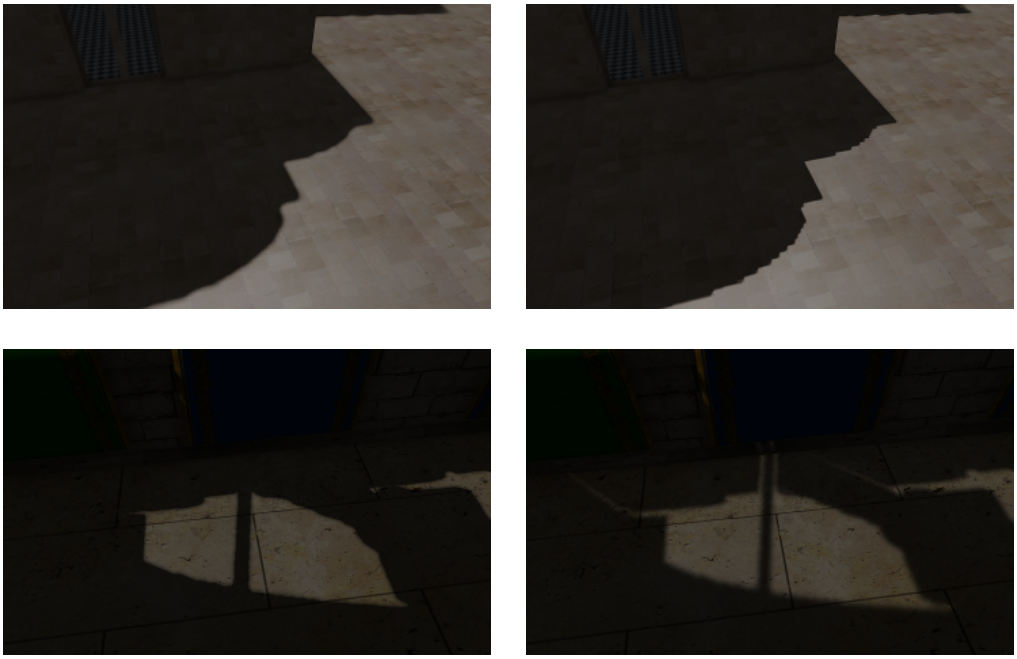

Figure 4: Upper row: Cascaded shadow maps, 256x256 texture, with and without filtering by a 3x3 Gaussian blur. Lower row: A threshold value of $\tau = 0.5$, on the left, masks the contours of a distant flagpole leaking through on the right.

The basic idea is to use moving shadow cameras to render a set of relatively small shadow maps. This can provide detailed shadows close to the eye, and cheap, low-quality, shadows further away. There exist many variants, but for this implementation, the work by Zhang et al. [2009] is useful. Mainly used as a tool for describing important, and general, techniques for flicker reduction and filtering across splits, the basic system "splits" the view frustum by covering more and more of it via a set of expanding bounding boxes.

By processing these bounding boxes, nesting them and ensuring uniform dimensions, they could be used as the building blocks for cascaded LPVs. Unfortunately, the cascaded version of LPVs have not been implemented within the time frame of work on this thesis, but I believe that the working CSM system should fulfill the prerequisites well.

## 4.3 Ambient Occlusion

Due to the low-frequency nature of LPVs, surface details of indirectly illuminated geometry must be achieved through ambient occlusion. The AO

buffers in figure 5 are the result of combining the original SSAO technique with another screen-space technique covered in a tutorial by Méndez [2010].

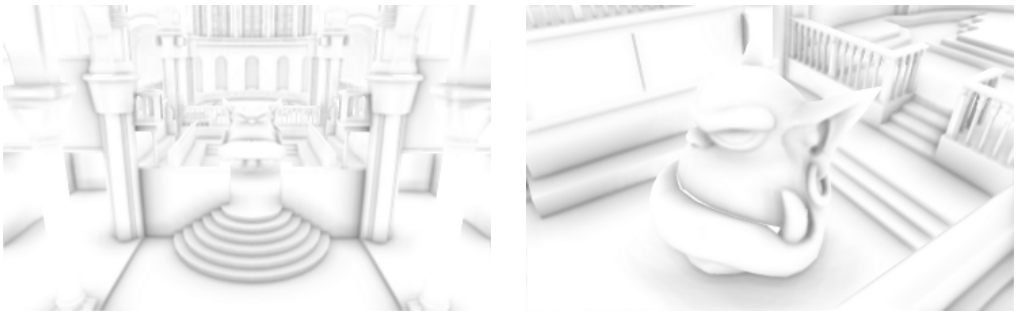

Figure 5: Ambient occlusion calculated in screen space.

The sampling artifacts of SSAO are more easily filtered, but the other method avoids self-occlusion and bright edges by using the following to compute the amount of occlusion per sample:

```
saturate(dot(normal, dir) - bias) * (1.0f / (1.0 + d * d))
```

The amount of occlusion by a potential occluder depends on the cosine of the angle between the occluded surface normal and direction towards the occluder, and is inversely dependent on the (squared) distance between the surfaces.

Subtle self-occlusion may still occur, which is why a small bias is subtracted from the dot product. Occluders that lie further away than a user-defined radius are ignored entirely. As the technique works in screen-space, it suffers from the usual problems of disappearing geometry, as is evident in figure 6.

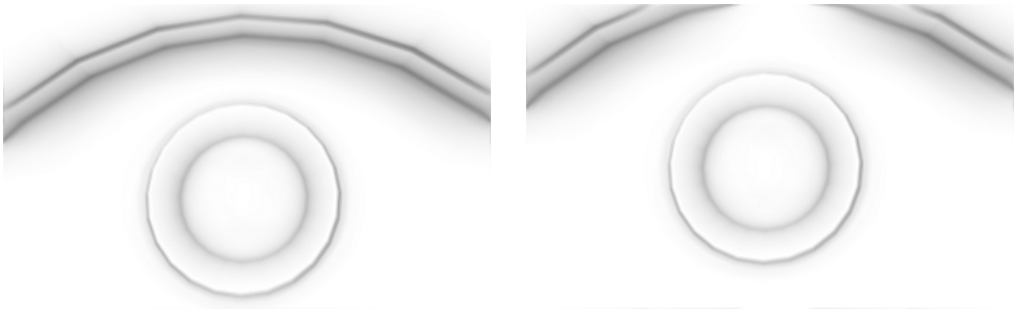

Figure 6: As the ceiling disappears from view, the wall underneath suddenly appears brighter.

## 4.4 Light Propagation Volumes

The first step of the LPV algorithm is to create the reflective shadow maps. Early on, storing the moments for variance shadow mapping was performed in the same pass as storing positions, normals and flux.

Various data have different requirements in the way of storage capacity, however, and the pixel format for each surface of the multiple render target should preferably be the smallest possible. Regrettably, adding surfaces of different bit depths to a MRT does not seem possible with OGRE's release candidate for version 1.8[9].

As the main light type for which implementation has focused on has been the spot light, and that type generally requires a higher resolution shadow map than what is appropriate for LPV injection, the current solution is to decouple the standard shadow mapping from RSM creation. While the redundant pass and render target switching is unfortunate, it allows for less costly pixel formats, and also very small RSMs (128x128, or even 64x64).

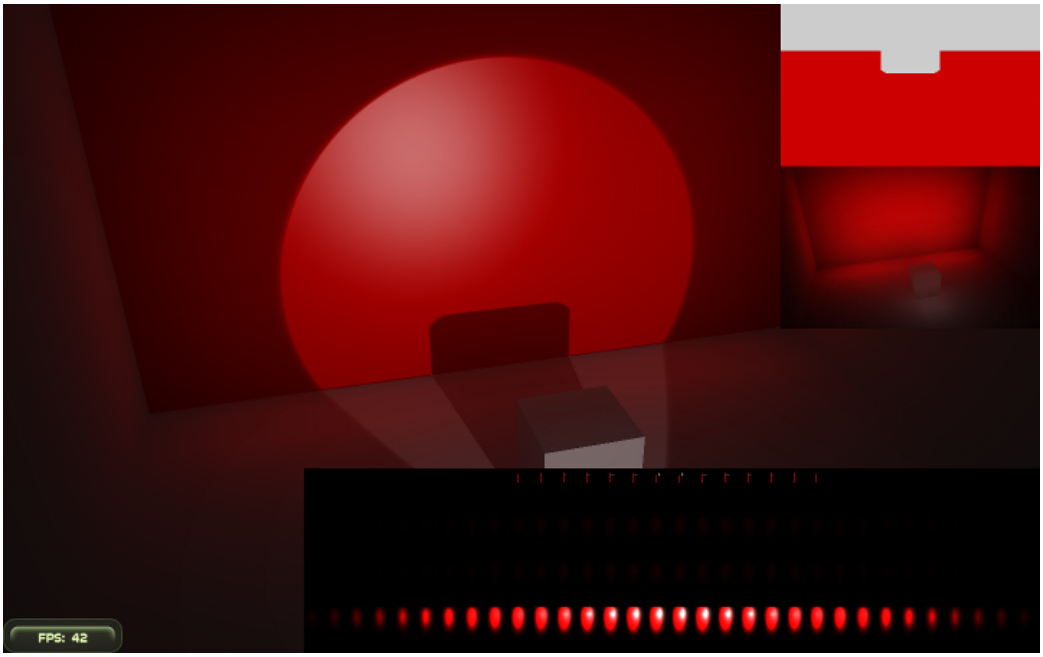


Figure 7: Early version of the LPV implementation.

[9]This is likely due to some bug, as recent OGRE versions should support this feature.

### 4.4.1 Injection

For injection of the LPV, a point cloud consisting of RSM width×height single vertices is rendered. The idea is to use the vertex ID to obtain texture coordinates for the RSM, fetch the RSM values in the vertex shader and pass them on to the geometry shader. Geometry shaders can output which depth layer of a volume texture the fragment shader should write to, which is necessary here, as the depth is unknown until the position has been fetched from the RSM.

Even though geometry shaders setting the correct layer can be compiled without complaint by the NVIDIA Cg compiler — and then used in an OGRE application — only a single layer will be written to. OGRE does not create volume textures as a single render target. Instead, each depth layer becomes a render target, and switching between them must be done as per usual on the CPU side.

As that will not work for injection, a 2D texture atlas is created, with depth layers side by side. This works well enough, and the same strategy is adapted for geometry injection. Extra care must be taken with boundary values when injecting geometry from the main camera:

```
cell = (worldPos - minCrnr) / (maxCrnr - minCrnr) * LPVsize;
cell -= 0.5f; // LPV centers are GV corners

if (outsideLPV(cell))
  clipSpacePosition = float4(-1000, 0, 0, 1);
else {
  cell = floor(cell); // NB extremely important!
  index2D.x = cell.x + cell.z * LPVsize;
  index2D.y = cell.y;
  ...
}
```

In the additional pass to combine the geometry volumes of the RSM and main camera, the result of that operation can be written to a proper volume texture.

Because each spot light has its own LPV and GV, the camera-visible geometry is injected from a downsampled buffer once, and then shared among the lights. The LPV size for each light is taken from the bounding box of the light volume used for deferred shading, slightly expanded to allow for propagation.

### 4.4.2 Propagation

The propagation is done in typical ping-pong fashion, where two sets of buffers alternate between inputs and render target. Preferably, the accumulation buffers should be bound as surfaces to each of the MRTs, and additive blending enabled only for those buffers. As current OGRE versions do not support this, however, a separate accumulation pass must be performed per propagation iteration.

Since all propagation is performed in world space, the SH projections needed for deciding how much flux is propagated towards each face, as well as those for re-projection, are pre-computed and uploaded to the GPU once every application run. The correct solid angles, which depend on both the current neighbor and face, are also sent as an array to avoid as much dynamic branching in the shader as possible.

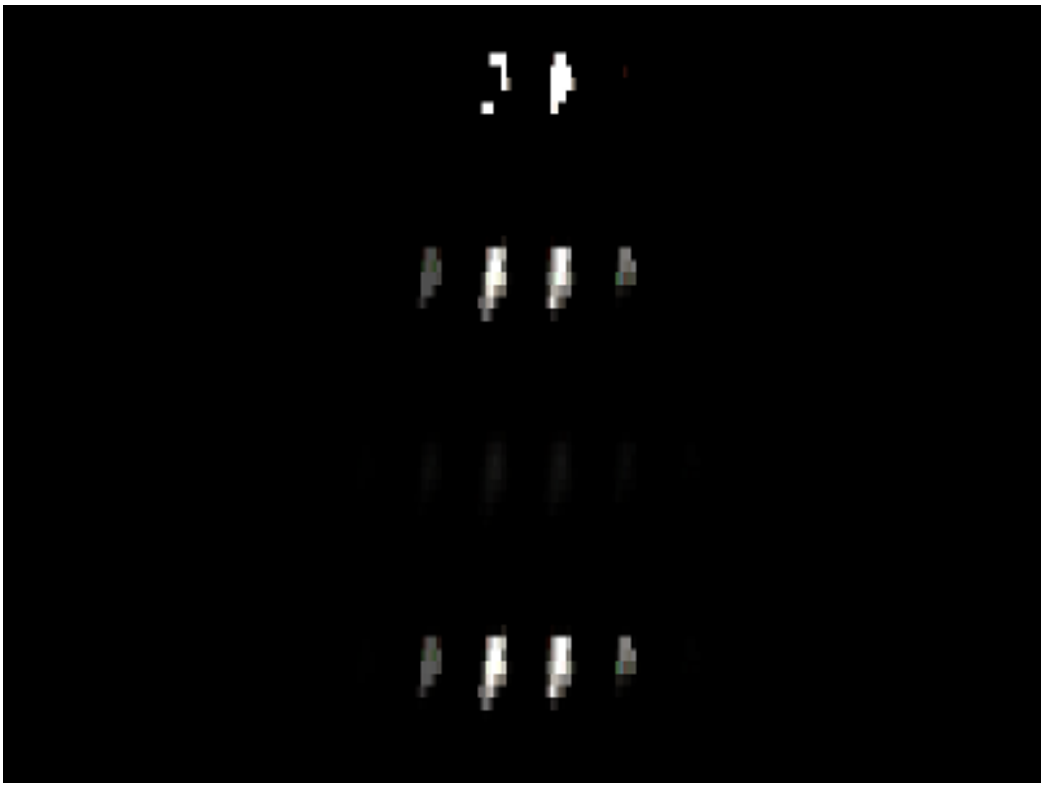

Figure 8: Light propagation without amplification, after 2 iterations.

The most expensive part of the propagation is the occlusion calculation. For a given voxel, the occlusion values for all 8 of its corners are fetched at the beginning of the shader program. Depending on which neighbor cell the

light propagation is currently gathered from, the 4 appropriate values are linearly interpolated.

In [Kaplanyan and Dachsbacher, 2010], it is said of the suggested propagation scheme that "[the] process conserves energy [as] is expected from light propagation in vacuum." However, when propagating light using the scheme, as described, the situation depicted in figure 8 arises.

The image consists of four unwrapped volume textures with depth layers placed horizontally. From top to bottom, it shows the injection stage, first propagation, second propagation, and accumulation. After the second propagation, only very little light remains to be propagated in the next iteration.

To counteract the intensity drop, the propagation shader amplifies the flux reflected onto each face by a user-defined parameter. In the middle of figure 9, we see the seventh and eight propagation step using an amplification factor of 8.0.

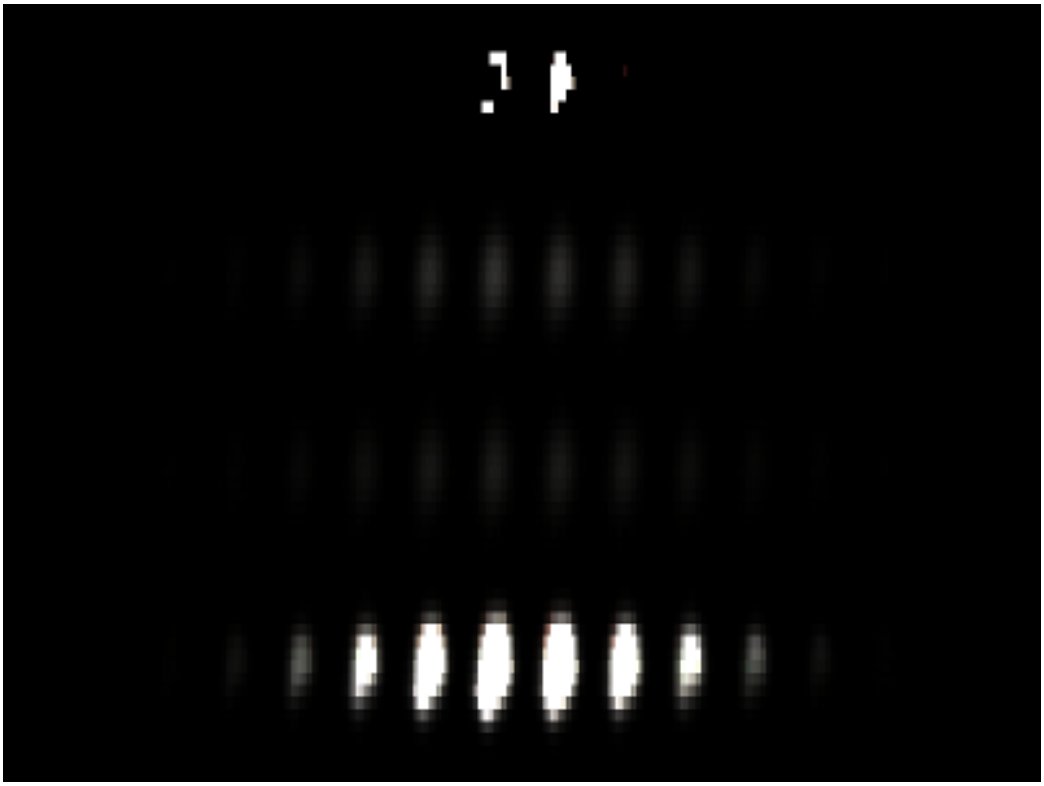

Figure 9: Light propagation with 8× amplification, after 8 iterations.

While the inclusion of an amplification factor has the benefit of acting as a parameter for artistic control, its necessity may unfortunately be an indicator of some subtle bug in the current implementation. (Note that figures 8 and 9

are used for visualization purposes only, and are not indicative of how the accumulated light will actually illuminate the scene.)

### 4.4.3 Lighting

The accumulated light is tri-linearly interpolated at each surface fragment visible from the eye. Using 2D accumulation buffers, this interpolation must be done manually to handle all three directions. In total, 3×3 texture fetches are required to acquire the interpolated SH coefficients for each color channel. For dampening of the of the incoming radiance based on the directional derivative, an additional 2×3×3 texture fetches must be made.

In addition to the many texture samples needed, the manual interpolation across depth layers was initially not satisfactory. The current approach is to render the 2D textures into a volume texture, similar to what is done for the geometry volumes.

For LPV dimensions like 24×24×24, or 32×32×32, the incurred overhead generally seems to compare favorably to the original method. Since each depth layer must be rendered separately, the excessive render target switching involved with larger dimensions makes the approach counterproductive, however.

The directional derivative is calculated as described in section 3. Taking the dot product of each SH coefficients vector with its derivative is used to see if they are deviating. Subsequent dampening is done as necessary by subtracting the (user-scaled) derivative vector from the sampled one[10]. Results can be seen in figure 10.

To convert the sampled data into irradiance, the negative of the surface fragment's normal is first projected into SH coefficients. This vector is dotted with the sampled coefficients for each color channel, and the resultant intensity is assumed to originate half a cell size in world units ($s_h$) away from the surface. The intensity is scaled by dividing with ${s_h}^2$ and then interpreted as incident radiance. Since we are dealing with low-frequency lighting, the radiance can in turn be interpreted as the surface irradiance [Tatarchuk, 2005].

[10]This is an ad-hoc method that seems to work fairly well in most cases. Values for scaling are usually between 1.0 and 3.0, but larger values may result in color artifacts.

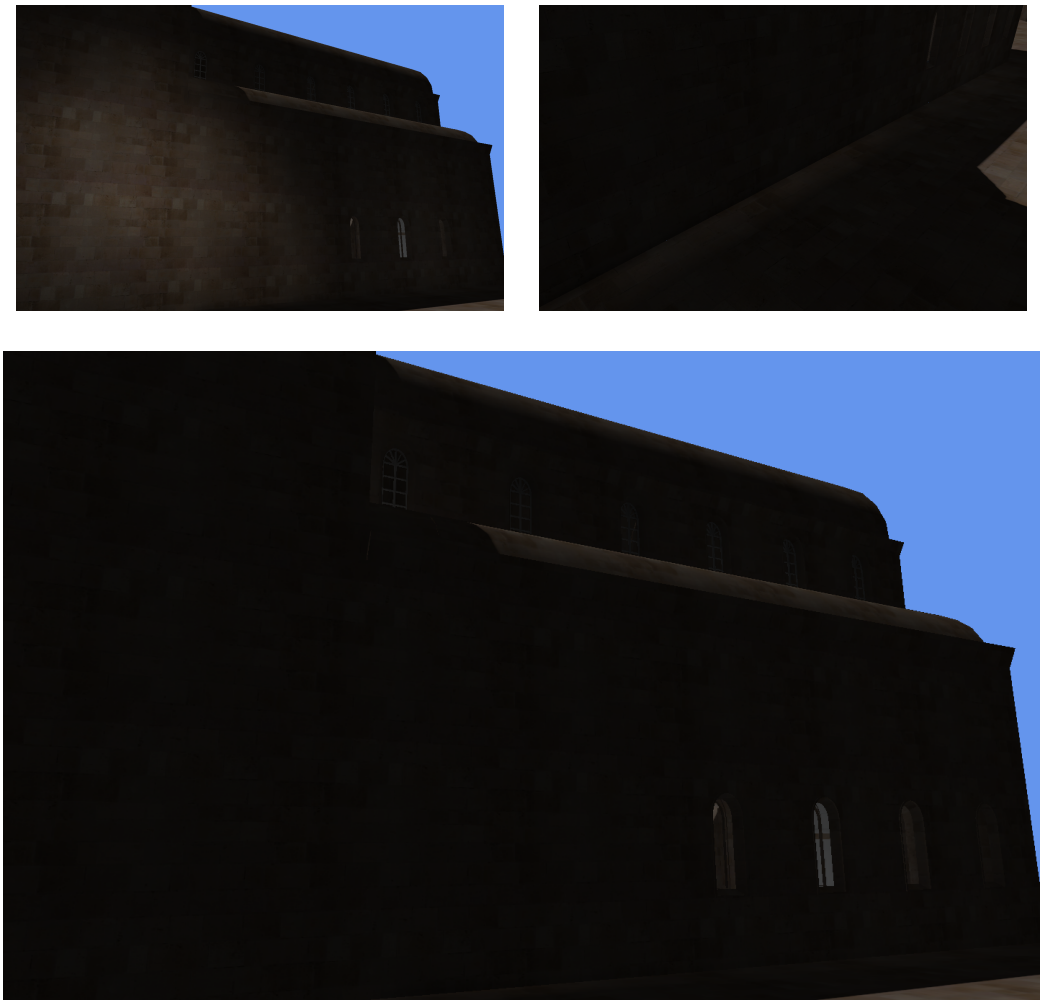

Figure 10: Top left: Lighting inside the building leaks through the wall. Bottom: Dampening removes most of the leakage. Top Right: Even after dampening, corners are still sensitive to the discretization involved in propagation.

# 5 Results and Discussion

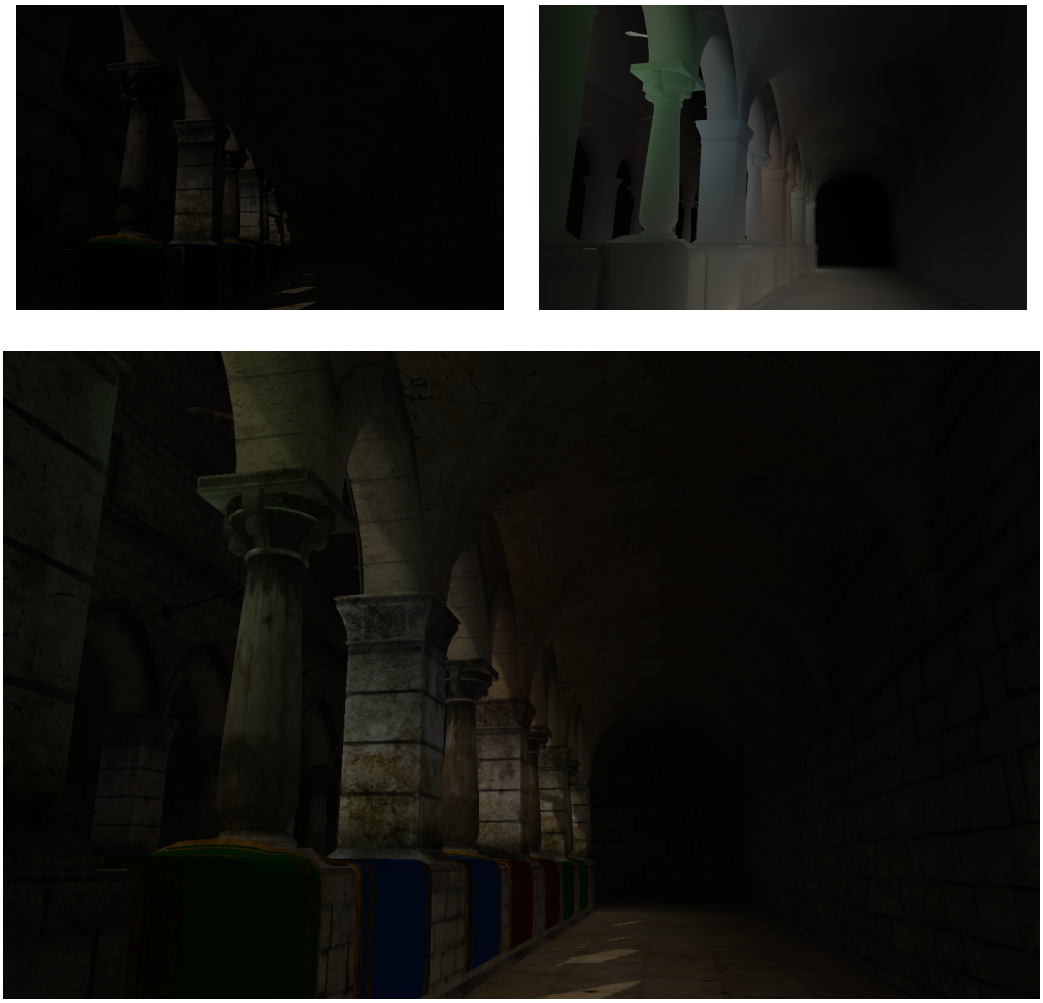

Figure 11: Top left: Direct lighting. Top right: Indirect illumination only. Bottom: Sponza atrium lit by direct and indirect illumination.

Figure 11 is a good example of the advantages of global illumination. The top left corner is a very sparsely lit view, the largest share of the screen almost black. Using a brighter, constant, ambient term here would really highlight the artificial nature of that approach. Adding ambient occlusion would bring out details in the background, and help the viewer sort out the spatial relationships of the unlit areas, but no more.

With the indirect illumination added, the image is more dynamic. The end of the corridor and the arched ceiling are still dark, since most of the light falls on objects close to the camera, and the floor. Color bleeding from

the banners gives each column its own appearance, and adds atmosphere to the frame.

## 5.1 Results

The implementation described in section 4 has almost exclusively been developed on a PC laptop (Intel i7 processor with 4 cores @ 2.20GHz, 8GB RAM) with an NVIDIA Geforce GT555M (2GB video memory) running the 64-bit version of Windows 7. Most work has been done using a release candidate (RC1) of OGRE 1.8.0 (but also the previous major release 1.7.4). All shaders were written using NVIDIA's Cg programming language.

Had the current implementation kept exclusively to using 2D atlases for all volumes, the full implementation would have worked using either one of OGRE's OpenGL or Direct3D 9 render systems. The latter, unfortunately, cannot handle volume texture render targets. Cascaded shadow mapping requires slightly different shader code for the two APIs, but otherwise, everything apart from LPVs works for both APIs without modifications.

The OGRE version used, has incomplete support for Direct3D (10 and) 11. Most importantly, MRTs are unsupported, which makes the render system currently unusable for this project.

For the complete LPV extension, OpenGL is therefore the only supported render system. Besides some unsupported features, like the mentioned problems with volume rendering[11] and less-than-optimal pixel formats for MRTs, using the gDebugger profiler reveals other issues. Rendering 11 frames of the scene depicted on the front page, 23.4% of the OpenGL calls use deprecated functions, 37.90% of the calls represent *redundant state changes*. Hopefully, future versions of OGRE will address some of these problems. Also, the currently developed gl3plus project seems like an interesting alternative to OGRE's existing OpenGL support.

The image on the front page of this thesis is rendered at a resolution of 1440×900, using a single 32×32×32 LPV and 32 propagation iterations. For that number of iterations, the scene renders at 58 FPS. In figure 12, we see that the number of iterations can be lowered without much loss of light, but with a significant increase in frame rate. For reference, figure 13 is rendered at 106 FPS with direct lighting only.

For a more far-spread propagation, either the number of iterations can be increased, or the LPV cell dimension can be decreased (so that each cell covers more world space). The most suitable combination of the two

[11] In [Kaplanyan et al., 2011], it is explicitly noted that "[i]t is important to note that the hardware capability to render into a volume texture tremendously improves performance."

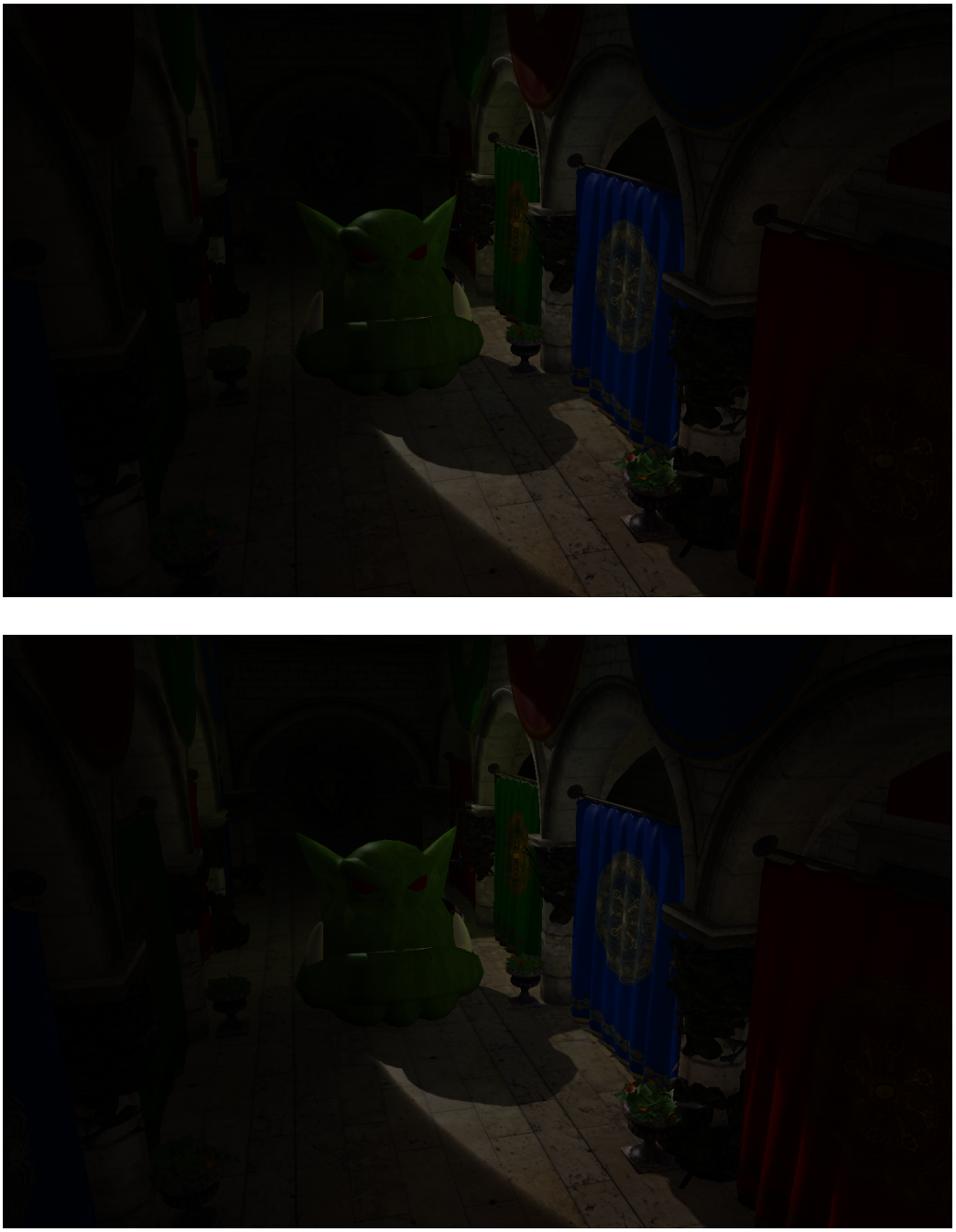

Figure 12: Top: 8 propagations, 73 FPS. Bottom: 16 propagations, 68 FPS. 32×32×32 LPVs are used in both cases, and the images are rendered at a resolution of 1440×900.

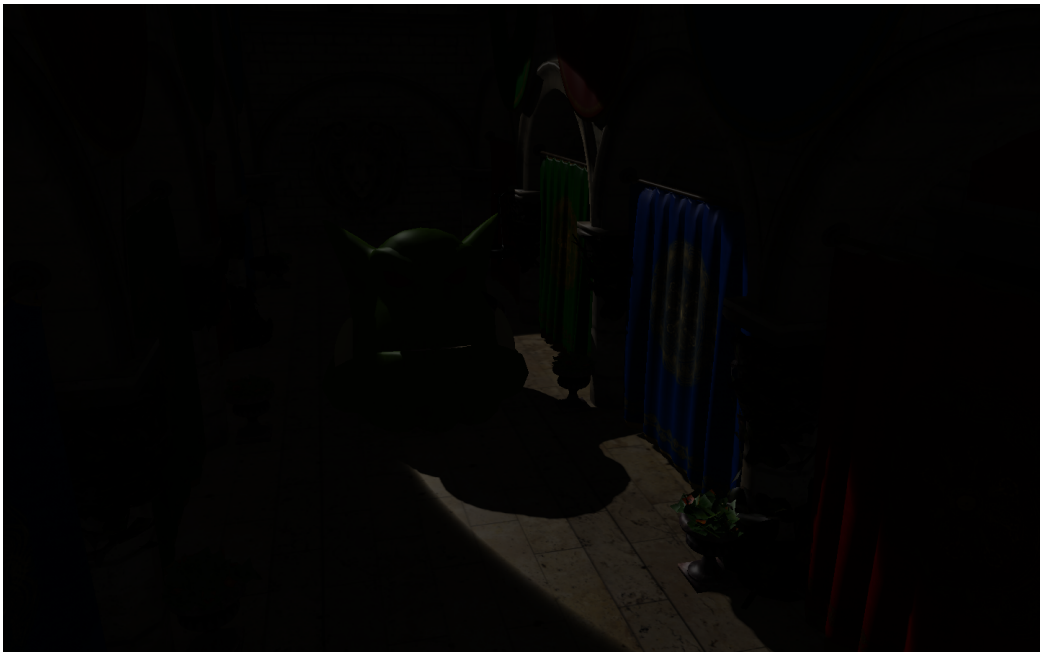

Figure 13: The scene on the first page, and in figure 12, rendered at 106 FPS with direct lighting only.

approaches depends on the desired effect and the specific scene.

Due to the low-order SH projections used, the inexact evaluation will result in some slight back-propagation. In combination with the propagation amplification needed to combat the intensity drop (section 4.4.2), increasing the number of propagations will exacerbate the back-propagation effect (brighter areas close to the reflective surfaces). Decreasing LPV dimensions, on the other hand, introduces more blur in the indirect lighting, and makes the fuzzy occlusion scheme even more approximate. Figure 14 shows an example of the Sibenik cathedral being illuminated with three spot lights.

## 5.2 Discussion

There are a number of parameters associated with LPVs. The amplification parameter used during propagation has been previously mentioned, and a similar parameter also exists for occlusion. In addition, the relationship between LPV dimensions and the world space dimensions of the corresponding bounding box, affects the range and detail of the indirect illumination. For achieving the best results, all these parameters should be fine-tuned for each particular scene.

The occlusion is, of course, based on a very rough approximation of the

Figure 14: Using three spot lights to light the Sibenik cathedral. Final images rendered in 49 FPS at a resolution of 1440×900. 16×16×16 LPVs, 16 propagations. (Note that the indirect illumination is calculated in both images, and its contribution toggled off/on, hence the identical frame rates.)

visibility term. It improves the visual quality by reducing the amount of light that would otherwise leak through objects, but does not result in detailed shadowing from the indirect illumination. Fortunately, due to the low-frequency aspect of diffuse indirect illumination, errors are hard to pinpoint, and may well be interpreted as the result of further bounces in some cases.

Adding main camera geometry heightens the effect of occlusion. But as with all screen-space effects, it suffers from having geometry moving in and out of view as the camera moves. In certain cases, this has a visual impact similar to that of the eyes adjusting to surrounding lighting conditions. In other cases, though, as objects occluding light that would otherwise propagate relatively far move in and out of view, the result is flickering in the indirect illumination. Figure 15 shows the effect varying occlusion.

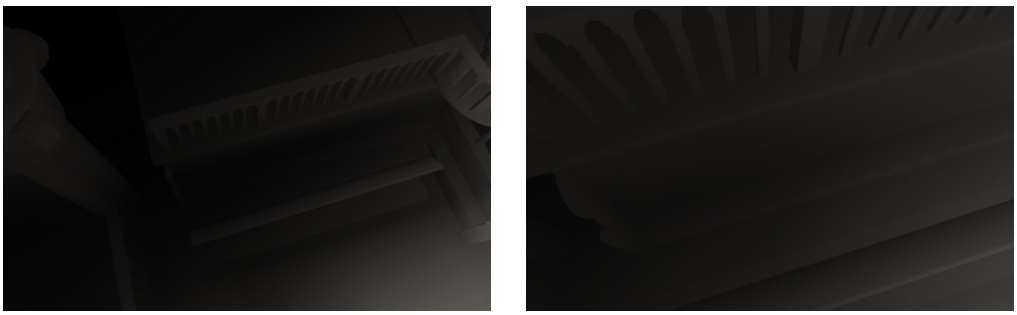

Figure 15: Fuzzy occlusion varies as objects move out of main view.

One of the truly positive feature of LPVs, however, is that any scene — except for some adjustment of the parameters mentioned above — will be

directly compatible with the technique. Not only does this mean that any current content pipeline for 3D models is unaffected, but all legacy models are supported as well.

### 5.2.1 Future Work

Cascaded LPVs have the highest priority for future work on the algorithm. Since directional lighting is such an important, and common, lighting technique, extending the current LPV implementation to handle it well would be a very important step in making the GI algorithm as complete as possible.

The following two paragraphs describe additions mentioned in [Kaplanyan and Dachsbacher, 2010] and [Kaplanyan et al., 2011]:

Depth peeling can be used to create denser geometry volumes, but will impact the frame rate. The diffuse nature of the indirect illumination allows LPVs to be rendered every few frames (every fourth in this implementation), and then possibly substituting the new result in a smooth manner. While the details for this substitution are largely omitted, one can posit that by using three accumulation volumes, temporal interpolation between the previous two should smooth flickering due to varying occlusion.

A very interesting addition to the original algorithm are glossy reflections. This involves ray marching through the LPV and averaging the contributions.

Various general additions and optimizations would also be useful. Ambient Occlusion Volumes is an interesting AO technique. Deferred shading suffers from being incompatible with hardware capabilities for anti-aliasing, and transparent/translucent objects must be rendered separately.

The current implementation is designed purely as an extension of OGRE's core functionality. Once it is integrated with the larger software system it is intended for, further evaluation must be done to see where optimization and possible additions are needed.

# 6 Conclusion

OGRE provides a unified API with extensive functionality for a number of diverse, but equally important, tasks: mathematical operations, scene management, low-level API calls, and asset import and handling. While all of those have generally simplified a lot of the work involved, there have also been occasions on which the rendering engine have been less suitable for the task at hand.

Troubleshooting various unexpected behavior can certainly be frustrating. But at the same time, it is a natural part of working with any large software system. More disheartening, then, are the instances where no amount of error search can help the fact that a wanted feature is simply not offered. In the case of the volume rendering, especially, it imposes an artificial limit on the efficiency of the LPV algorithm.

Nonetheless, the current LPV extension works relatively well, and seems likely to scale nicely with future hardware. In addition to that, OGRE is continuously developed. As some of the more common strategies for real-time GI in general, and LPVs in particular, become standard (e.g., shadow map MRTs, volume rendering), OGRE will likely obtain the necessary features. The fact that the engine is open source also means that it is possible for *me* to contribute to the process of incorporating them.

# References

Akenine-Möller, T., E. Haines, and N. Hoffman (2008). *Real-Time Rendering 3rd Edition*. Natick, MA, USA: A. K. Peters, Ltd.

Bunnell, M. (2005). Dynamic Ambient Occlusion and Indirect Lighting. In M. Pharr and R. Fernando (Eds.), *GPU Gems 2*, Chapter 14, pp. 223–233. Addison-Wesley Professional.

Crow, F. C. (1977, July). Shadow Algorithms for Computer Graphics. *SIGGRAPH Comput. Graph. 11*(2), 242–248.

Dachsbacher, C. and M. Stamminger (2005). Reflective Shadow Maps. In *Proceedings of the 2005 symposium on Interactive 3D graphics and games*, I3D '05, New York, NY, USA, pp. 203–231. ACM.

Dachsbacher, C. and M. Stamminger (2006). Splatting Indirect Illumination. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, I3D '06, New York, NY, USA, pp. 93–100. ACM.

Greger, G., P. Shirley, P. M. Hubbard, and D. P. Greenberg (1998, April). The Irradiance Volume. *IEEE Computer Graphics and Applications 18*(2), 32–43.

Grosch, T. and T. Ritschel (2010). Screen-Space Directional Occlusion. In W. Engel (Ed.), *GPU Pro: Advanced Rendering Techniques*, Chapter 4.2, pp. 215–230. A K Peters.

Kajalin, V. (2009). Screen-Space Ambient Occlusion. In W. Engel (Ed.), *ShaderX7: Advanced Rendering Techniques*, Chapter 6.1, pp. 413–424. Charles River Media.

Kajiya, J. T. (1986). The Rendering Equation. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '86, New York, NY, USA, pp. 143–150. ACM.

Kaplanyan, A. and C. Dachsbacher (2010). 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, New York, NY, USA, pp. 99–107. ACM.

Kaplanyan, A., W. Engel, and C. Dachsbacher (2011). Diffuse Global Illumination with Temporally Coherent Light Propagation Volumes. In W. Engel (Ed.), *GPU Pro 2*, Chapter 3.5, pp. 185–203. A K Peters/CRC Press.

Keller, A. (1997). Instant Radiosity. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '97, New York, NY, USA, pp. 49–56. ACM Press/Addison-Wesley Publishing Co.

Koonce, R. (2007). Deferred Shading in Tabula Rasa. In H. Nguyen (Ed.), *GPU Gems 3*, Chapter 19, pp. 429–457. Addison-Wesley Professional.

Laine, S., H. Saransaari, J. Kontkanen, J. Lehtinen, and T. Aila (2007). Incremental Instant Radiosity for Real-Time Indirect Illumination. In *Proceedings of Eurographics Symposium on Rendering*, Grenoble, France, pp. 277–286. Eurographics Association.

Landis, H. (2002). Production-Ready Global Illumination. *Siggraph Course Notes 16*(3), 2002.

Lauritzen, A. (2007). Summed-Area Variance Shadow Maps. In H. Nguyen (Ed.), *GPU Gems 3*, Chapter 8, pp. 157–182. Addison-Wesley Professional.

McGuire, M. (2010). Ambient Occlusion Volumes. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, Aire-la-Ville, Switzerland, Switzerland, pp. 47–56. Eurographics Association.

McGuire, M. and D. Luebke (2009). Hardware-Accelerated Global Illumination by Image Space Photon Mapping. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, New York, NY, USA, pp. 77–89. ACM.

Méndez, J. M. (2010). A Simple and Practical Approach to SSAO. Website. http://www.gamedev.net/page/resources/_/technical/graphics-programming-and-theory/a-simple-and-practical-approach-to-ssao-r2753. Accessed: 27/05/2012.

Mittring, M. (2007). Finding Next Gen: CryEngine 2. In *ACM SIGGRAPH 2007 courses*, SIGGRAPH '07, New York, NY, USA, pp. 97–121. ACM.

Nichols, G. and C. Wyman (2009). Multiresolution Splatting for Indirect Illumination. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, I3D '09, New York, NY, USA, pp. 83–90. ACM.

Nijasure, M., S. Pattanaik, and V. Goel (2005). Real-Time Global Illumination on GPUs. *Journal of Graphics, GPU, & Game Tools 10*(2), 55–71.

Papaioannou, G. (2011). Real-Time Diffuse Global Illumination Using Radiance Hints. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, New York, NY, USA, pp. 15–24. ACM.

Pettineo, M. (2010). Position From Depth 3: Back In The Habit. Website. http://mynameismjp.wordpress.com/2010/09/05/position-from-depth-3/.
Accessed: 27/05/2012.

Pranckevičius, A. (2009). Compact Normal Storage for small G-Buffers. Website. http://aras-p.info/texts/CompactNormalStorage.html.
Accessed: 27/05/2012.

Ramamoorthi, R. and P. Hanrahan (2001). An Efficient Representation for Irradiance Environment Maps. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '01, New York, NY, USA, pp. 497–500. ACM.

Ritschel, T., E. Eisemann, I. Ha, J. D. K. Kim, and H. Seidel (2011, December). Making Imperfect Shadow Maps View-Adaptive: High-Quality Global Illumination in Large Dynamic Scenes. *Computer Graphics Forum 30*(8), 2258–2269.

Ritschel, T., T. Grosch, M. H. Kim, H. Seidel, C. Dachsbacher, and J. Kautz (2008, December). Imperfect Shadow Maps for Efficient Computation of Indirect Illumination. *ACM Trans. Graph. 27*(5), 129:1–129:8.

Shishkovtsov, O. (2005). Deferred Shading in S.T.A.L.K.E.R. In M. Pharr and R. Fernando (Eds.), *GPU Gems 2*, Chapter 9, pp. 143–166. Addison-Wesley Professional.

Sloan, P. (2008). Stupid Spherical Harmonics (SH) Tricks. Presentation. Game Developers Conference (GDC '08), San Francisco, CA.

Sloan, P., J. Kautz, and J. Snyder (2002, July). Precomputed Radiance Transfer for Real-Time Rendering in Dynamic, Low-Frequency Lighting Environments. *ACM Trans. Graph. 21*(3), 527–536.

Tabellion, E. and A. Lamorlette (2004). An Approximate Global Illumination System for Computer Generated Films. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, New York, NY, USA, pp. 469–476. ACM.

Tatarchuk, N. (2005). Irradiance Volumes for Games. Website. http://developer.amd.com/media/gpu_assets/Tatarchuk_Irradiance_

Volumes.pdf.
Accessed: 28/05/2012.

Tatarchuk, N., H. Chen, A. Evans, A. Kaplanyan, J. Moore, D. Jeffries, J. Yang, and W. Engel (2009). Advances in Real-Time Rendering in 3D Graphics and Games. In *ACM SIGGRAPH 2009 Courses*, SIGGRAPH '09, New York, NY, USA. ACM.

Thiedemann, S., N. Henrich, T. Grosch, and S. Müller (2011). Voxel-Based Global Illumination. In *Symposium on Interactive 3D Graphics and Games*, I3D '11, New York, NY, USA, pp. 103–110. ACM.

Torus Knot Software Ltd (2000–2011). OGRE — Open Source 3D Graphics Engine. Software. http://www.ogre3d.org/.

van Antwerpen, D. (2011). Improving SIMD Efficiency for Parallel Monte Carlo Light Transport on the GPU. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, New York, NY, USA, pp. 41–50. ACM.

Williams, L. (1978, August). Casting Curved Shadows on Curved Surfaces. *SIGGRAPH Comput. Graph. 12*(3), 270–274.

Zhang, F., A. Zaprjagaev, and A. Bentham (2009). Practical Cascaded Shadow Maps. In W. Engel (Ed.), *ShaderX7: Advanced Rendering Techniques*, Chapter 4.1, pp. 305–329. Charles River Media.