Real-time Rendering Using Layered Depth Maps

Frederik Peter Aalund* — DTU — Student (s093279)

Jeppe Revall Frisvad† — DTU — Supervisor

Jakob Andreas Bærentzen‡ — DTU — Supervisor

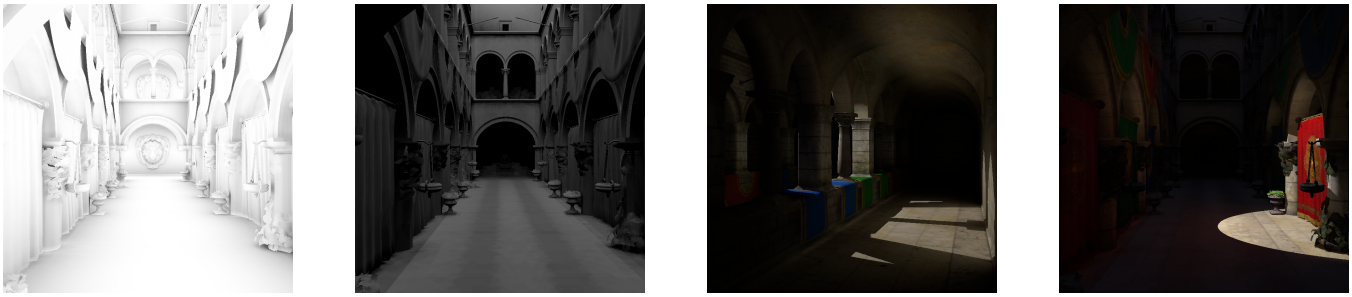


Figure 1: *All images are generated using rasterization and layered depth maps. From left to right: Ambient obscurance, ambient occlusion, single-bounce indirect lighting, and environment lighting combined with indirect lighting.*

Abstract

A layered depth map is an extension to the well-known depth map used in rasterization. Multiple layered depth maps can be used as a coarse scene representation. We develop two global illumination methods which use said scene representation. The first is a real-time ambient occlusion method. The second is an interactive single-bounce indirect lighting method based on photon differentials. All of this is implemented in a rasterization-based pipeline.

Keywords: real-time rendering, layered depth maps, order-independent transparency, global illumination, ambient occlusion, indirect lighting, photon differentials

Links: WEB CODE

Acknowledgements

I want to thank my supervisors, Jeppe Revall Frisvad and Jakob Andreas Bærentzen, for their guidance and support throughout this project. Moreover, for their prompt and detailed replies to my e-mails and for giving me access to the neccessary hardware. I would also like to thank Christian Viggo Larsen and Alessandro Dal Corso for helping me with shader debugging. Lastly, I want to thank Alessandro Dal Corso for letting me use his computer to collect results.

*e-mail: frederikaalund+ldm2015@gmail.com

†e-mail: jerf@dtu.dk

‡e-mail: janba@dtu.dk

Contents

1 Introduction

In this section, we will first provide motivation for our research topic. First, we give a full description of the project on which this report is based. Second, we outline the prerequisites for reading this document. Third, we provide an overview of the remaining sections.

1.1 Motivation

Rasterization is a popular real-time rendering technique based on *primitive traversal* of the scene geometry [Akenine-Möller et al. 2008]. That is, each primitive is rasterized into pixels individually and the sum of all primitive contributions constitutes the rendered image (Figure 2a). Contrast this technique to *ray-tracing* which is based on *pixel traversal* [Whitted 1980]. That is, each pixel's color is computed by ray-tracing from the pixel's position on the image plane through the primitives that constitute the scene geometry (Figure 2b).

Despite the many similarities between rasterization and ray-tracing, the traversal order is a distinctive difference. Rasterization only requires *local* scene information (a single primitive) each step. Ray-tracing, however, requires *global* scene information (all primitives) each step. Consequently, rasterization has a smaller memory footprint for complex scenes. This is a key advantage of rasterization and it has lead to dedicated acceleration through GPUs [Akenine-Möller et al. 2008]. Simultaneously, this is a significant limitation of rasterization since only local information can be used in shading.

Overcoming the local limitation of rasterization in real-time rendering has been the focus of recent research (see Section 2). Common for all methods is the use of an auxiliary data structure which contains a coarse representation of the scene geometry. Programmable GPU features such as *fragment shaders* are adapted to construct said data structures in real-time.

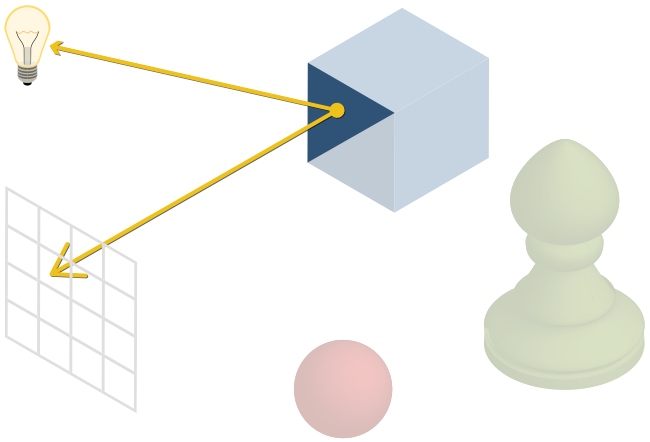
During rasterization, a *shader* can then query the auxiliary data structure for global scene information. The latter can be used to implement *global illumination* and thus overcome the local limitation of rasterization. This is an ideal combination of the performance characteristics of rasterization with the physical correctness of global illumination (Figure 2c).

1.2 The Project

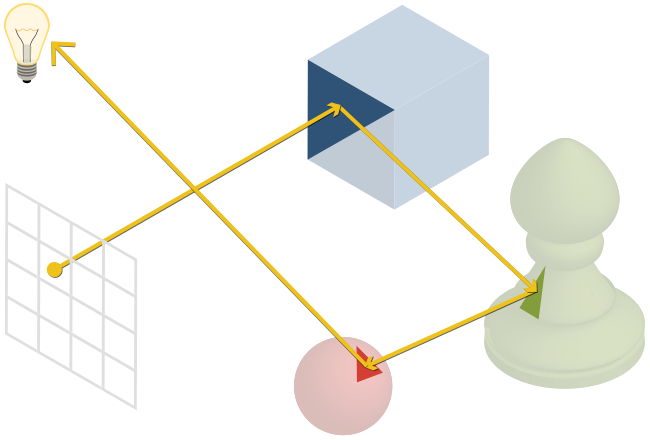
We use an auxiliary data structure based on *layered depth maps* that can be used to query global scene information during rasterization as done in [Krüger et al. 2006; Bürger et al. 2007; Zhang et al. 2008; Niessner et al. 2010; Hu et al. 2014]. Layered, in the sense that all *depth values* (not just a single one) are stored in the map. The novelty in our approach is that each layered depth map is pre-sorted which in turn allows for a fast tracing algorithm.

We also present two global illumination methods which use rasterization in combination with our auxiliary data structure: *ambient occlusion* (AO) and single-bounce indirect lighting. These methods are meant to demonstrate the applicability of our auxiliary data structure. We use a path traced reference to evaluate the image quality of our results. Furthermore, we compare the AO implementation with a screen-space approach.

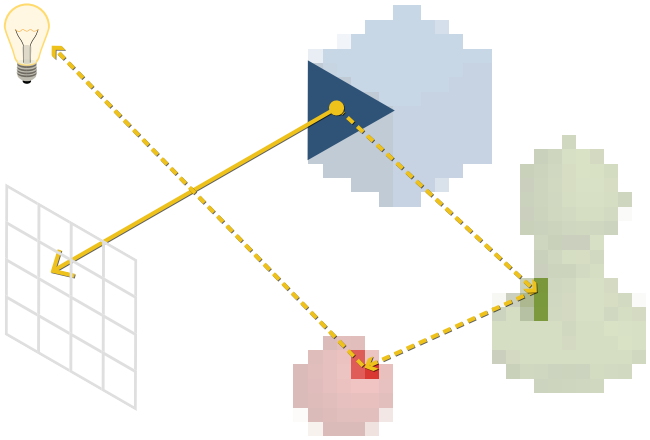
The auxiliary data structure and the accompanying global illumination techniques will be implemented in a C++14/1z application using OpenGL 4.x for hardware acceleration. The target platforms will be desktops or laptops with a recent GPU running Windows, OS X, or a Unix-like operating system. Note that the presented



(a) *Rasterization. Each cell in the grid represents a pixel on the image plane (4×4 in this case). The triangle from the blue cube is projected into the image plane (primitive $\rightarrow$ pixels). This happens in isolation of the rest of the scene. Consequently, only local illumination can be used in shading. On the other hand, the memory requirements are low since only a single primitive is rendered at a time.*



(b) *Ray-tracing. Tracing starts from the image plane and can reflect between multiple surfaces before ending at a light source (pixel $\rightarrow$ primitives). Hence a ray-tracer can implement global illumination. On the other hand, the memory requirements are high since all primitives must be kept in memory.*



(c) *Rasterization with an auxiliary data structure. Global scene information can now be queried (dashed arrows) from an approximate scene representation (pixelated objects).*

Figure 2: *Comparison of rendering strategies.*

techniques are not limited to said platforms; we are merely targeting them in order to demonstrate our results.

1.3 Prerequisites

We assume that the reader is familiar with the basic concepts and methods of real-time rendering and physically-based rendering. We will, however, provide explanations of advanced topics as they become relevant. Likewise, we will provide details about the underlying hardware when the hardware either constrains or guides our approach. This includes elaborations on performance characteristics that are tied to our choice of GPU.

Specifically, we will not explain the purpose of a fragment shader or the derivation of the rendering equation. Please refer to [Akenine-Möller et al. 2008] and [Pharr and Humphreys 2004] for the basics of real-time rendering and physically-based rendering, respectively.

1.4 Overview

This remainder of this report is divided into four parts: A cursory part followed by three in-depth main parts. First, we give a brief overview of scene representations used in real-time rendering. Second, we go into details with layered depth maps. Third, we introduce a real-time AO technique to demonstrate the use of our auxiliary data structure. Fourth, we introduce an indirect lighting technique based on photon differentials which also uses our auxiliary data structure.

The three main parts will present background theory, previous work, method design, and implementation details. As mentioned in Section 1.3, most basic concepts are assumed known. Consequently, the given background theory is merely intended to establish notation and provide general historical context. The latter is done through an analysis of previous work. We generally present the previous work chronologically but reserve the right to deviate from the timeline when appropriate for interjections. Then, we use the result of the analysis to design a method suitable for our use case. Lastly, we go into the implementation details that make our design usable in practice.

The three main parts are followed by our results and findings. We will provide a qualitative comparison of image quality (correctness) as well as a quantitative comparison of performance. Then we discuss our results and suggest improvements to our approach and design. Lastly, we conclude the project and propose topics for future study.

2 Overview of Scene Representations

This section will give a brief overview of various auxiliary scene representations used to provide global information during rasterization. The discussion is limited to representations that have been used in real-time rendering. The purpose is to put our chosen method, layered depth maps, in a larger context. As such, we will not go into detail with any of the alternatives but merely mention them. This section can be skipped if you are already familiar with the topic.

Depth Map One of the earlier approaches is to use the *depth map* from the rasterization pipeline as a coarse scene representation. This can be used to approximate AO [Mittring 2007; Shanmugam and Arikan 2007], reflections (so-called *image-space reflections*) [Kasyan et al. 2011], and even single-bounce indirect lighting [Ritschel et al. 2009]. The depth map is often used together with a normal map, diffuse reflectance map, and other maps common in a deferred rendering pipeline.

Multi-view All deferred maps are screen-space limited (no scene information outside the field of view). One remedy is to simply render the scene from multiple views in order to get more geometric information [Ritschel et al. 2009]. However, multi-view rendering adds additional overhead to the rendering pipeline. If only depth values are needed, then the lights' shadow maps can be reused as a view source [Vardis et al. 2013]. This mitigates some of the overhead.

Multi-resolution Performance can be further improved by using a multi-resolution map [Nichols and Wyman 2009]. That is, a map which embeds coarser versions of itself. This approach is similar to mipmapping but uses a conservative min-max reduction scheme instead of linear interpolation.

Shadow Map The shadow map alone can be augmented with additional scene information (positions, normals, radiant flux) to produce a *reflective shadow map*. The latter is used to approximate single-bounce indirect lighting [Dachsbacher and Stamminger 2005]. The indirect diffuse light is often low-frequency which enables the use of low-resolution and approximate shadow maps (known as *imperfect shadow maps*) [Ritschel et al. 2008].

Layered Depth Map The limitations of the depth map led to the use of a layered depth map [Ritschel et al. 2009]. As with depth maps, multi-view solutions are often used [Ritschel et al. 2009; Niessner et al. 2010; Tokuyoshi and Ogaki 2012b]. Through time, applications have varied from simple reflections [Zhang et al. 2008] to bidirectional path tracing via rasterization [Tokuyoshi and Ogaki 2012b]. Layered depth maps are explored further in Section 3.

Voxel Grid Another approach is to compute a *dynamic sparse voxel octree* during primitive traversal. The octree serves as a coarse scene representation and can be queried efficiently with voxel cone tracing to approximate ambient occlusion and glossy reflections [Crassin et al. 2011]. Similarly, a voxel structure can be used to store potential ray hits in a *ray grid*. The ray grid is then used to combine rasterization and ray-tracing in a hybrid technique [Zirr et al. 2014].

Surfel Cloud Each primitive is tessellated into *surface elements* (surfels) and shading effects are applied via splatting [Nalbach et al. 2014]. A surfel is really an oriented disk which augments the properties of its parent primitive. Contrast this to the traditional pipeline where primitives are rasterized directly and shading effects are applied via gathering. E.g., as done in *screen-space ambient occlusion* (SSAO) [Mittring 2007; Shanmugam and Arikan 2007]. With surfels, all geometric information sent through the rasterization pipeline is retained. I.e., no information is lost due to occlusion.

Hybrids A coarse voxel grid can be combined with the precision of layered depth maps in a hybrid technique [Hu et al. 2014]. The voxel grid is first queried to find a conservative depth interval. The latter is then refined by a range-limited lookup into a layered depth map. Another approach is to combine reflective shadow maps with layered depth maps [Tokuyoshi and Ogaki 2012b]. Yet another method uses the coarse voxel grid for visibility queries and determines the indirect light using reflective shadow maps [Sugihara et al. 2014].

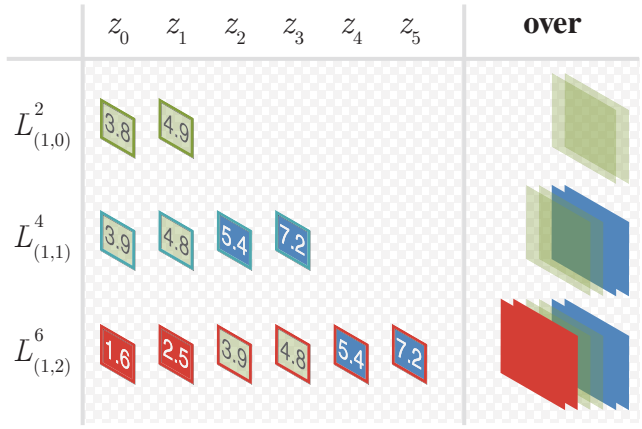


Figure 3: *L^k_p for the three pixels of Figure 4b. I.e., for $p = (1,0)$, $p = (1,1)$, and $p = (1,2)$. Note that k may very between pixels. The largest k denotes the number of layers (six in this case). A sequence, L^k_p, does not necessarily contain a value for each layer. E.g., $L^2_{(0,1)}$ which only has two entries. Also shown is* **over** *applied to each fragment color in sequence.*

3 Layered Depth Maps

Layered depth maps were invented to solve the issue of rendering transparent objects with a rasterization pipeline [Maule et al. 2011]. However, our motivation to study layered depth maps is not related to transparency. We use layered depth maps as a coarse scene representation in order to provide global information during rasterization. Nevertheless, previous work on layered depth maps in the context of transparency can be readily applied to our use case. Therefore, we devote the next couple of sections to study the traditional use of layered depth maps.

First, the background section will explain the original motivation for layered depth maps. Next, previous work is explored. The design section evaluates the previous work and selects a suitable method. Lastly, the implementation section describes how to implement a layered depth map in practice.

3.1 Background

This section will explain the origin of depth maps, the problems they solve, and their limitations. Said limitations motivate layered depth maps which are explained next.

3.1.1 Depth Map

A depth map stores a single depth value per pixel. The depth of a pixel is the distance from the viewer to the first surface represented by the pixel [Catmull 1974] (Figure 4a).

Most graphics hardware incorporate a depth map into its rasterization pipeline [Akenine-Möller et al. 2008]. The depth map is also referred to as the *z-buffer* or *depth buffer* in this context because the z-coordinate in *normalized device coordinates* (NDC) denotes the distance into the screen[1]. The z-buffer is used to resolve fragment visibility so that the closest fragments (lowest *z-value*) are drawn on top. For this purpose, it's only interesting to store the closest depth (lowest z-value) in the depth map. Therefore, only a single value per pixel is needed.

A single depth value, however, is not enough to handle *transparency*.

Transparency Color in real-time rendering is usually represented by an RGBA tuple[2], were A is the *alpha* value (or *opacity*) in the range $[0;1]$. Opacity is the complement of transparency. Translucency being the general case, transparency is the special case of non-scattering light passing through an object. It's an oft-used simplification in real-time rendering.

Typically, two colors are *blended* into a single by color as if one color is in front of the other. This process is represented by the Porter-Duff **over** operator [Porter and Duff 1984]

$$\textbf{over} : (\text{RGBA}, \text{RGBA}) \rightarrow \text{RGBA}$$

which can be implemented as

$$\mathbf{f}\,\textbf{over}\,\mathbf{b} = \begin{bmatrix} \mathbf{f}_\text{A}\mathbf{f}_\text{R} + (1-\mathbf{f}_\text{A})\,\mathbf{b}_\text{R} \\ \mathbf{f}_\text{A}\mathbf{f}_\text{G} + (1-\mathbf{f}_\text{A})\,\mathbf{b}_\text{G} \\ \mathbf{f}_\text{A}\mathbf{f}_\text{B} + (1-\mathbf{f}_\text{A})\,\mathbf{b}_\text{B} \\ \mathbf{f}_\text{A} + (1-\mathbf{f}_\text{A})\,\mathbf{b}_\text{A} \end{bmatrix}$$

where $\mathbf{f}$ is the *front* color and $\mathbf{b}$ is the *back* color [Akenine-Möller et al. 2008]. The subscripts denote the color components. In practice, a slightly different definition is used

$$\mathbf{f}\,\textbf{over}\,\mathbf{b} = \begin{bmatrix} \mathbf{f}_\text{R'} + (1-\mathbf{f}_\text{A})\,\mathbf{b}_\text{R} \\ \mathbf{f}_\text{G'} + (1-\mathbf{f}_\text{A})\,\mathbf{b}_\text{G} \\ \mathbf{f}_\text{B'} + (1-\mathbf{f}_\text{A})\,\mathbf{b}_\text{B} \\ \mathbf{f}_\text{A} + (1-\mathbf{f}_\text{A})\,\mathbf{b}_\text{A} \end{bmatrix}$$

where $\mathbf{f}$ is stored using *pre-multiplied alphas*

$$\begin{bmatrix} \text{R'} \\ \text{G'} \\ \text{B'} \\ \text{A} \end{bmatrix} = \begin{bmatrix} \text{A} \cdot \text{R} \\ \text{A} \cdot \text{G} \\ \text{A} \cdot \text{B} \\ \text{A} \end{bmatrix}$$

Besides being more efficient (one less multiplication), colors with pre-multiplied alphas can be linearly interpolated (as done during texture filtering) which is why this approach is favored in practice [McDonald 2013].

Note that **over** is *non*-commutative since

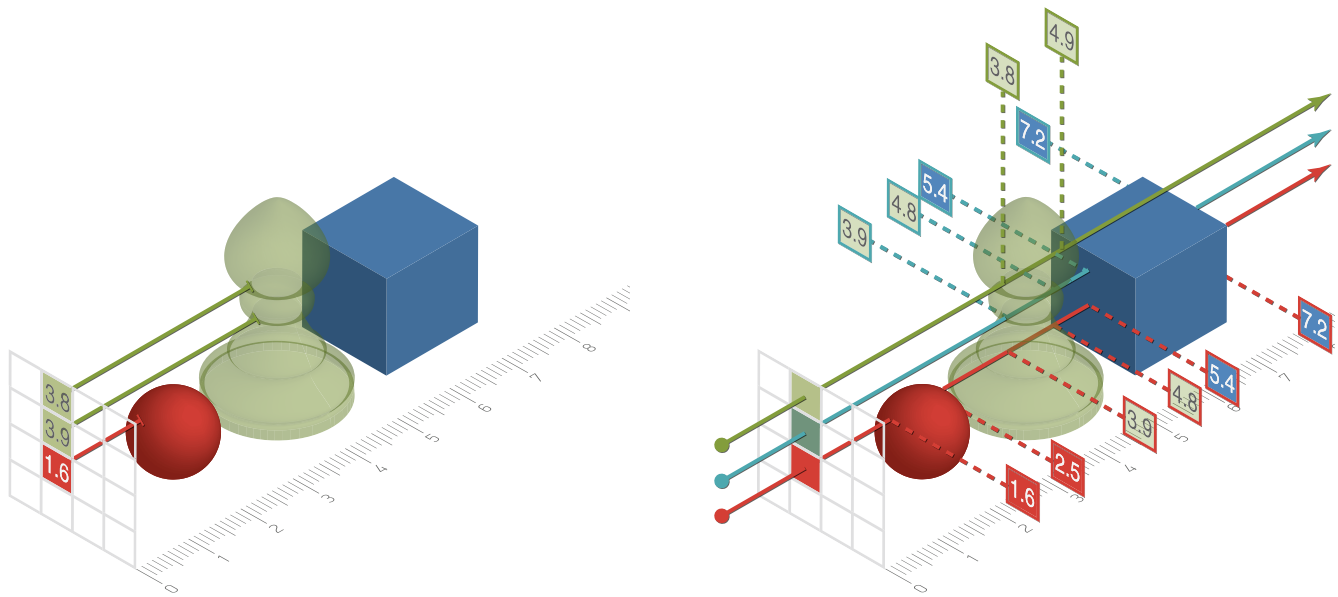
$$\mathbf{f}\,\textbf{over}\,\mathbf{b} \neq \mathbf{b}\,\textbf{over}\,\mathbf{f}$$

for some $\mathbf{f}, \mathbf{b} \in \text{RGBA}$ (except when $\mathbf{f} = \mathbf{b}$). This is is true whether alphas have been pre-multiplied or not. In other words, **over** is order-dependent. Consequently, all the fragments of a pixel must be blended in depth-sorted order to correctly resolve transparency. The depth map cannot be used for this purpose as it only stores a single depth value (the closest). This limits the depth map to *opaque* surfaces[3].

[1] The original term z-buffer was coined by its inventor [Catmull 1974]. Some authors capitalize the term to Z-buffer [Akenine-Möller et al. 2008]. We prefer the term depth map as it's consistent with the term layered depth map.

[2] A diffuse/specular texture usually contains tuples of wavelength-banded reflectivity coefficients. The color of a texture is really an interpretation of said coefficients under ideal conditions.

[3] An incomplete solution is to depth-sort the primitives before rasterization [Govindaraju et al. 2004]. The problem is overlap cycles between primitives. When an overlap cycle occurs, depth-sorting is not applicable and the solution fails [Maule et al. 2011; Knowles et al. 2012].



(a) *Depth map. Each pixel stores the distance from the viewer to the first geometric intersection along the view ray of said pixel. Here, three view rays are shown along with the depth value of the closest fragment. Orthographic projection is used but the principle is the same for perspective projection.*

(b) *Layered depth map. Each pixel stores multiple depth values. Here, three view rays are shown. The depth values are given in the colored squares along each view ray. Both front and back faces are rasterized. Note that multiple fragments may map to the same pixel.*

Figure 4: *Storage of depth values.*

3.1.2 Layered Depth Map

A layered depth map stores multiple depth values per pixel. More precisely, to each pixel, p, is associated a sorted sequence of depth value, $L_p^k = (z_0, z_1, \ldots, z_k)$, where $z_0 \leq z_1 \leq \cdots \leq z_k$ for some k. A depth value, z_l, is said to be *in* the lth *layer* (and the lth layer is said to *contain* z_l). This way, layers enforce *relative depth ordering* between any two pixels. Within a pixel, each depth value is uniquely identified by its layer. We may omit k and write L_p to denote that the sequence is not of fixed length. See Figures 3 and 4b for reference.

Note that while all depth values are in a layer, a layer is not necessarily associated with all pixels. That is, layers can vary in *size* (the number of depth values in a given layer). This is the case in Figure 3.

A layered depth map will usually hold some additional fragment data besides the depth value. E.g., an RGBA tuple denoting the fragment's surface color. Therefore, we will sometimes refer to L_p^k as a sequence of fragments. We have omitted the additional fragment data from the formal description since the following discussion is focused on depth values.

Note that a depth map is a special case of a layered depth map with only a single layer that contains all depth values.

Transparency Layered depth maps succeed where depth maps fail: They can handle transparent surfaces. **over** is simply applied sequentially to the depth-sorted elements of each L_p sequence. Compare Figure 4a and Figure 4b to see the difference. As such, the graphics pipeline may render primitives in *any* order and rely on the data structure behind L_p to sort the fragments according to depth. This is referred to as *order-independent transparency* (OIT). The difficult part is to choose a suitable data structure for L_p that fits into a rasterization-based pipeline. Even more so to choose one that is efficient enough for real-time purposes. Section 3.2 will go into detail on how this is achieved.

Terminology We distinguish between the notion of a layered depth map and its implementation. The former is defined by L_p^k and the latter is one of the various X-buffers which will be presented next. In previous work, notion and implementation have sometimes been covered by the same term. Notably, the so-called A-buffer which was both the first to introduce layered depth maps and a corresponding implementation. Consequently, the A-buffer has become synonymous for both.

To confuse matters more, the layered depth map notion also has different names. E.g., the *multi-layer z-buffer* [Max et al. 1996], *layered depth images* [Gortler et al. 1997], *fragment lists* [Szécsi and Illés 2012], the *layered fragment buffer* [Knowles et al. 2012], etc. We will use the term layered depth map for the rest of this report.

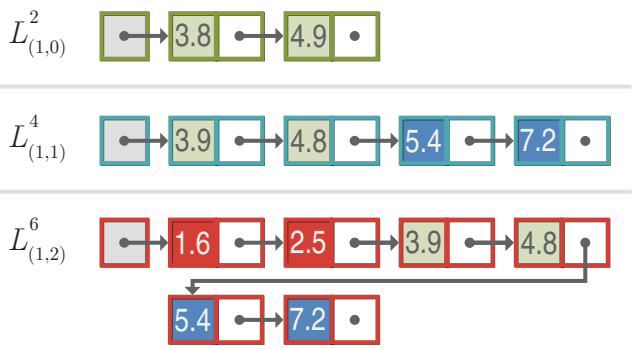


Figure 5: *A-buffer. Each L_p^k sequence is stored in a singly linked list. Note the extra spaced used for the next pointer (arrow). A head pointer is used to mark the start of the list. A null pointer (grey dot) marks the end of the list.*

3.2 Previous Work

This section is a comparative study of various layered depth map implementations. From this study we chose a candidate which we will use for the remainder of the report.

3.2.1 A-buffer

Layered depth maps were introduced with the *anti-aliased, area-averaged, accumulation buffer* (A-buffer) [Carpenter 1984] as a depth map replacement that can handle transparency. This was before the emergence of GPUs and the given implementation is meant for offline use in the REYES system. Despite the the differences between the REYES system and say the OpenGL pipeline, the A-buffer is still relevant today. The anti-aliased, area-averaged, accumulation buffer is named so because the original implementation also handles anti-aliasing and weighs color by the sub-pixel surface area.

The authors describe what is essentially a per-pixel singly linked list of fragments sorted according to depth (Figure 5). Thus the A-buffer is the first to introduce the notion of L_p and even suggests a data structure for it. The sorting algorithm is left unspecified, however. Likewise, there is no discussion on the memory bounds though the authors do suggest a C struct layout for the nodes in the linked list. See Figure 8 for a possible memory layout.

The authors also propose various transparency-specific optimizations. E.g., skipping fragments behind opaque surfaces. Our use case is to preserve geometric information so in our case such optimizations are irrelevant.

3.2.2 Z^3

The Z^3 data structure seeks to improve on the anti-aliasing of the A-buffer by storing not only the depth value, z, but also the slopes (derivatives) of the depth value, z_x and z_y, in the x- and y-direction, respectively [Jouppi and Chang 1999]. Hence the name Z^3 since it stores three z-related values. Furthermore, the authors suggest to store a constant k fragments per pixel. That is, to use an L_p^k sequence (Figure 6a). If a pixel has more than k fragments, then two existing fragments are merged to make room for the new fragment.

Unfortunately, a fixed k makes the Z^3 approximate since it can potentially discard geometric detail (through the merging of fragments). As such, it is not useful for our purpose of creating a scene representation. Still, Z^3 is the first to suggest a memory-bound data structure (since k is kept fixed). Specifically, it is suggested to use contiguous storage (Figure 6b). This may lead to wasted space if some layers are sparse. Furthermore, Z^3 is proposed as a hardware extension and it is unclear which rasterization system it is meant to integrate with (if any).

3.2.3 Hardware Proposals

In the wake of Z^3, the years 2000–2003 saw many proposals which implement OIT with hardware extensions. The first GPUs of the Nvidia GeForce and the ATI Radeon[4] lines had just come out in 1999 and 2000, respectively [Mark and Proudfoot 2001]. GPU architectures were still young. However, none of these proposals have been exposed through either DirectX or OpenGL as part of commercially available hardware. Still, we mention some of them here for completeness.

R-buffer The *recirculating fragment buffer* (R-buffer) is a pointerless derivative of the A-buffer that provides OIT for a fixed-function pipeline (such as OpenGL 1.x) [Wittenbrink 2001]. The R-buffer is essentially a *first-in, first-out* (FIFO) buffer of the incoming fragments. It's pointerless, since a FIFO buffer can be implemented with contiguous storage. The authors also provide a two-pass algorithm meant to be implemented in hardware. In the first pass over the scene geometry, all transparent fragments are added to the R-buffer. In the second pass over the R-buffer, fragments are either blended into the *framebuffer* or put back (recirculated) in the R-buffer. Thus the second pass must be repeated until the R-buffer is empty.

The actual logic of the second pass is complex and out of scope of this report. The interesting part is that the fragments are stored in the order they are fed into the pipeline and not with regard to their x- and y-coordinates. In other words, the R-buffer provides unique storage for all fragments.

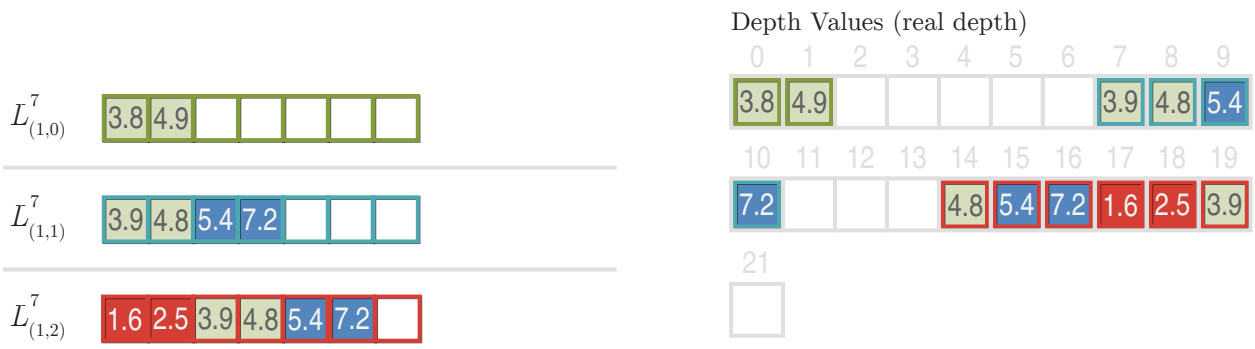
F-buffer The *fragment-stream buffer* (F-buffer) is a new *render target* (RT) that stores all incoming fragments in a FIFO buffer [Mark and Proudfoot 2001]. As such, it is very similar to the R-buffer. Special passes are used to empty the F-buffer and write directly to the framebuffer. The F-buffer is not only meant to solve the problem of order-independent transparency. It is a general proposal meant to be used with a programmable pipeline (such as OpenGL 2.x and above) in combination with any technique that requires a FIFO buffer of fragments.

The ATI 9800 and X800 actually shipped with a hardware F-buffer implementation [Houston et al. 2005]. However, it was never exposed through a publicly available API. The F-buffer has since then not been part of newer ATI/AMD GPUs' feature list.

The R-buffer and the F-buffer are some of the first to propose that unique storage should be provided for all fragments in the fixed-function and programmable pipeline, respectively. As will soon become apparent, this is a key concept that later methods reproduce via advanced features (OpenGL 4.x).

Others The simply-named *fragment buffer* stores per pixel linked lists (similar to the A-buffer) in special-purpose hardware [Lee and Kim 2000]. The *delay stream* is a FIFO (similar to the R-buffer) which stores deferred primitives while subsequent occlusion information is gathered [Aila et al. 2003]. OIT is implemented by querying the delay stream for primitive information.

[4] Now AMD Radeon.



(a) *Using the example seen in Figure 3. Shown here for $k = 7$. Each L_p^k sequence is stored in a fixed-length contiguous array. Note that a lot of space is wasted storing null values for pixels with few actual depth values.*

(b) *Memory layout. This figure uses the example seen in Figure 6a. Three pixels are rendered so the storage requirements are $k \times pixels = 7 \times 3 = 21$.*

Figure 6: Z^3.

There are also earlier OIT-related hardware proposals [Schilling and StraBer 1993; Winner et al. 1997]. However, they are unlikely to see a present-day implementation since the overall GPU hardware architecture has changed dramatically in the past 18–22 years.

3.2.4 k-buffer

A layered depth map with a fixed number of layers, $L_p^k = (z_1, z_2, \ldots, z_k)$ for some constant k, is called a k-buffer [Callahan et al. 2005]. Note that Z^3 already introduced this idea in [Jouppi and Chang 1999] but did not coin the term k-buffer. As we mentioned about Z^3, a limited depth value sequence, L_p^k, is not particularly useful to represent the scene geometry. Thus we will not go into too much detail with k-buffer but only elaborate on the technical innovations that accompanied them.

One k-buffer uses *multiple render targets* (MRT) to store the individual fragments [Callahan et al. 2005]. In a single pass, the fragments are written to the MRT, read back, sorted, and blended together. At the time, this allowed for $k = 7$ (6 from the MRT and one for the incoming fragment) in a single pass. Note that *read-modify-write* (RMW) from MRT in a single pass is undefined behaviour in OpenGL. The authors also note this but found that it worked in practice. Furthermore, they suggest to add memory objects with arbitrary read/write and synchronization primitives as extensions to OpenGL. Both features later became standard and are used by recent methods (Section 3.2.5).

Further advances in the available number of RTs allows for $k = 16$ [Bavoil et al. 2007]. Though this is accompanied with a special batching of primitives to mitigate the undefined behaviour of RMW.

Another implementation uses a multisample texture to store multiple depth values per pixel (instead of anti-aliasing samples) [Myers and Bavoil 2007]. This allows for $k = 8$ in a single pass. The implementation is standard compliant since it uses the stencil buffer to route depth value into the subpixels of the multisample texture.

Common for all approaches is that k is rather low. Still, it seems to produce good results when used for transparency. Furthermore, k can artificially be increased by using multiple passes. However this requires object sorting as well.

All in all, the early k-buffer implementations are not ideal for scene representation. However, they did inspire hardware innovations which allow us to use better algorithms today (and without undefined behaviour).

3.2.5 Hardware Advancements

The years 2008–2009 were quiet with regard to layered depth map implementations. Fortunately, the same time period saw significant hardware and API improvements. In the following technical aside, we describe the advancements which are related to layered depth maps.

SSBO The framebuffer provides a fixed amount of storage for each pixel [Segal et al. 2014]. Usually, the RT attached to the framebuffer stores RGBA tuples. MRT allows for more advanced usage (such as deferred shading and k-buffers) but can't store all incoming fragments in a complex scene. This is the primary reason why the various k-buffer implementations must use a fixed k.

The *shader storage buffer object* (SSBO) introduced as an OpenGL 4.x extension is essentially a contiguous chunk of memory that is shared between *shader invocation* [Brown et al. 2014a]. A shader invocation can RMW *any* memory location of an SSBO. Contrast this to the earlier OpenGL memory model which has always restricted writes to the shader invocation's pixel coordinates (and where RMW resulted in undefined behaviour).

With SSBOs it is now possible to store all fragments during rasterization (Figure 7). Unlike the framebuffer, however, an SSBO does not have any built-in synchronization between shader invocations. An SSBO is just a chunk of memory. All synchronization must be defined explicitly by the shader author.

Atomic Operations Fortunately, the SSBO extension also provides basic atomic operations which can be used to synchronize shader invocations. E.g.,

```
 int32_t atomicExchange(inout  int32_t a,  int32_t b)
uint32_t atomicExchange(inout uint32_t a, uint32_t b)
```

which atomically sets `a` to `b` and returns the old value of `a`.

Additionally, dedicated atomic counters introduced as another OpenGL 4.x extension provide an efficient way to atomically increment an integer from all shader invocations [Licea-Kane et al. 2012]. E.g.,

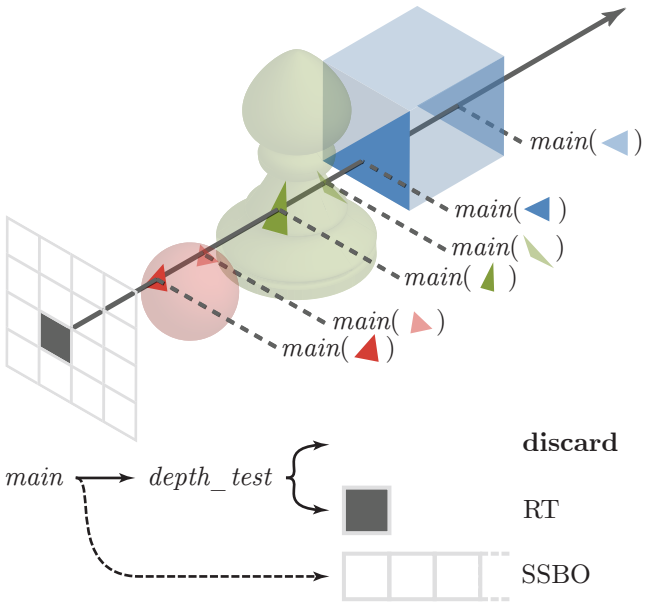


Figure 7: *Primitive processing in rasterization. Back-faces (desaturated primitives) are also rendered. Notice that multiple fragments (six in this case) may map to the same pixel (grey rectangle). Each fragment triggers a shader invocation (a call to main). Moreover, the shader invocations trigger in an undefined order so that any permutation is possible. Pre-OpenGL 4.3 (solid arrows), fragments are depth tested and either discarded or stored in the RT. Post-OpenGL 4.3 (dashed arrow), fragments can optionally be stored in an SSBO. Note that the SSBO can store multiple fragments.*

```
uint32_t atomicCounterIncrement(atomic_uint c)
```

which atomically increments `c` and returns the old value. Note the special `atomic_uint` type which must reference an external object (e.g., it can't be declared locally in a function).

Both of the aforementioned extensions are now both part of the OpenGL 4.3 Core Profile [Segal et al. 2013]. In conclusion, any GPU that conforms to the OpenGL 4.3 specification is *sufficient* to concurrently construct singly linked lists. Sufficient, because standard compliance does not guarantee a performant implementation.

For completeness it should be mentioned that there are equivalent primitives in DirectX [Yang and McKee 2010; Gruen and Thibieroz 2010; Thibieroz 2011]. Without loss of generality, we will use SSBOs for the remainder of this report.

3.2.6 Per-pixel Fixed-length Arrays (Z^3 revisited)

The above-mentioned advancements allowed researchers to revisit layered depth maps. One example is the use of an SSBO to store *per-pixel fixed-length arrays* (PPFLA) of depth values [Liu et al. 2009a; Liu et al. 2010; Crassin 2010a]. This approach is similar to Z^3 but only stores a single depth value (and not the two z-value derivatives) per fragment. Recall that Z^3 stores L_p^k sequences with a fixed k. As such, the memory layout is exactly that of Figure 6b. The method uses three passes:

1. **Clear.** A *count buffer* of per-pixel atomic counters, `count`, is reset to zero.
2. **Fragment Storing.** The scene geometry is rasterized and depth values are put into SSBO memory. The memory location can be determined from the fragment's xy-coordinates as `k * (y * width + x) + count++`. Here, `width` is the viewport's width and `k` is k. Note that the increment of `count` must be atomic (e.g., using `atomicAdd`).
3. **Sorting.** The fragments in each layer are depth-sorted using bubble sort.

The simplicity of the approach makes it very performant. The memory requirements depends on the size of the viewport and k. Overflow occurs if `count` becomes larger than k. The difficult part is to find a k which matches the scene's depth complexity. Conservative choices of k requires a lot of memory. Most of this memory will remain unused if the layers are sparse. Some authors therefore suggest a more memory-conservative approach [Crassin 2010b] (see Section 3.2.8).

3.2.7 Pre-sorted Per-pixel Fixed-length Arrays

The **Sorting** pass can be skipped by by depth-sorting during construction of the PPFLA [Liu et al. 2009a; Liu et al. 2010]. Said arrays are called *pre-sorted per-pixel fixed-length arrays* (PSPPFLA). The key is to use insertion sort implemented with OpenGL 4.x's `atomicMin` operation

```
int32_t atomicMin(inout int32_t a, int32_t b)
```

which atomically sets `a` to min(`a`,`b`) and returns the old value of `a`. With insert sort, only two passes are required:

- **Clear.** The per-pixel array entries are initially set to the maximum possible depth value, `max_depth`. `count` is cleared as before.
- **Fragment Storing.** An incoming fragment is stored by going linearly through all `count + 1` existing fragments (starting at `k * (y * width + x)`) and applying `incoming = atomicMin(existing, incoming)` where `existing` and `incoming` are the existing and incoming fragment, respectively. `count` is incremented as before.

All empty cells will contain `max_depth`. Initially, `existing` will contain `max_depth` and `incoming` will simply be stored. As the array grows, the **Fragment Storing** step will iterate from start to end and sort the array by ascending values. Because the array is only accessed with atomic operations, no race conditions can occur.

Serially, insertion sort algorithm has asymptotic complexity of $O(n^2)$ where n is the number of elements. Asymptotically faster sorting algorithms exist but insertion sort has the nice property that it can be applied in parallel during construction. Moreover, it can be implemented using only a single `atomicMin` instruction. These properties combined mitigate the otherwise poor asymptotic complexity.

The major downside to this approach is that the stored fragments are limited to 32–64 bits of data. This is because `atomicMin` only exists in 32-bit and 64-bit variants. Furthermore, the 64-bit `atomicMin` is a new addition to consumer hardware and not yet wildly adopted [Liu et al. 2010; Lefebvre et al. 2013]. Packing both the fragment's depth value and color into 32 bits causes artifacts due to the loss of precision [Liu et al. 2010]. However, if only the depth value needs to be stored (as in our use case), then 32 bits are sufficient.

While PPFLA and PSPPFLA are simple and fast, the memory requirements restrict their usefulness in practice. Furthermore, we aim to find a method that does not limit depth complexity. Therefore, we won't go into further details with PPFLA and PSPPFLA.

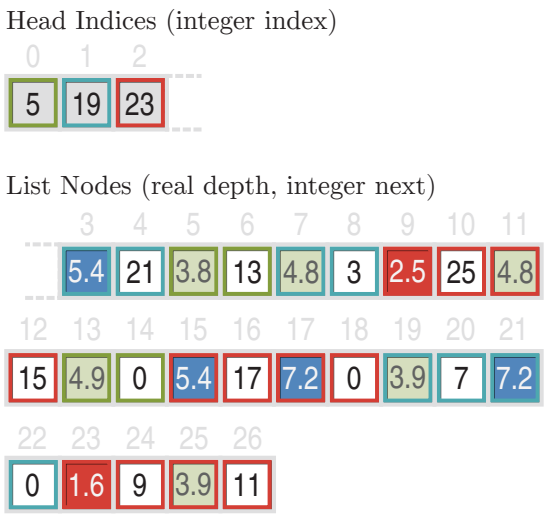


Figure 8: *PPSLL memory layout. This figure uses the example seen in Figure 5. Both head indices and list nodes are stored in the same contiguous chunk of memory. Note that this is only one permutation out of all list node orderings.*

3.2.8 Per-pixel Singly Linked Lists (A-buffer revisited)

The A-buffer also saw renewed interest. With SSBOs and the accompanying atomic operations, *per-pixel singly linked lists* (PPSLL) can be constructed [Yang et al. 2010; Yang and McKee 2010; Gruen and Thibieroz 2010; Thibieroz 2011].

Construction Three passes are required: **Clear**, **Fragment Storing**, and **Sorting**. The method itself is relatively simple and only spans a few lines of code. The difficult part is to arrange the various operations to avoid race conditions. Furthermore, to ensure unique memory allocation for each shader invocation from the SSBO.

We defer the complete technical explanation to Section 3.4.1. For now, we will highlight the attributes that can be directly compared to other methods. That is, the memory requirements and the sorting approach.

Memory The *OpenGL shading language* (GLSL) restricts SSBO access to array-like integer indexing. That is, pointer indirection is not supported[5]. Consequently, the list nodes must reference each other by integer indices. This is merely a technical inconvenience; the semantics are unaffected. The head and next pointers simply become integer indices.

The head indices are stored in one SSBO and the list nodes are stored in another SSBO [Yang et al. 2010]. Alternatively, both head indices and list nodes can be stored in the same SSBO [Lefebvre et al. 2014]. The head indices are then stored at the beginning and the list nodes follow directly afterwards (Figure 8). An index of 0 denotes the end of the list[6].

Recall that any permutation of shader invocations is possible (Figure 7). Consequently, any permutation of list nodes is possible (one such permutation is shown in Figure 8). Therefore, a list's nodes may be far apart in memory. E.g., the list of $L^4_{(1,1)}$ in Figure 8 whose nodes are at indices 19, 7, 3, and 21. When the lists are traversed, a node fetch is likely to trigger a cache miss which has a negative impact on performance. This is not an artifact of the GPU architecture; the very same behaviour can be observed with a CPU implementation. See Section 3.2.9 for a more memory-compact method.

While SSBOs can store vast amounts of data, they do have a fixed size. Unlike CPU memory allocation, an SSBO must be allocated before the shader is executed and can't be re-allocated during shader execution. Moreover, the scene's depth complexity is not known prior to the first rasterization run. Consequently, it is impossible to know how much memory to allocate for the SSBO beforehand. Thus the client application must detect overflow and allocate enough SSBO memory for the next frame [Yang et al. 2010]. As such, the memory requirements are unbounded. A monotonically increasing allocation scheme is one solution (similar to C++'s `std::vector`). With such a scheme, however, a sudden spike in depth complexity will reserve an otherwise unnecessary amount of memory. As of yet, there is no optimal solution.

In spite of all of the above-mentioned shortcomings, per-pixel linked lists require significantly less memory in practice than, say, per-pixel fixed-size arrays [Crassin 2010b]. Though this is not the case if all pixels have many fragments (as in Figures 6b and 8). Under practical circumstances, however, most pixels will have a small number of fragments and per-pixel linked lists are the better choice.

Sorting Recall that the list nodes are stored in arbitrary order. Consequently, a sorting pass is needed to depth-sort the lists before they actually conform to the L_p definition. First, the linked lists are copied into a shader-local array. Then, an arbitrary sort algorithm can be applied on the local array. Insertion sort ($O(n^2)$) and bitonic sort ($O(n \log^2 n)$) have been suggested by the original authors [Liu et al. 2010; Yang et al. 2010; Thibieroz 2011]. Shell sort ($O(n^2)$) was proposed later [Knowles et al. 2012]. As it turns out, insertion sort outperforms the aforementioned sort algorithms and even traditional $O(n \log n)$ sort algorithms (e.g., merge sort) for small n [Knowles et al. 2012]. Recently, a register-based block sort algorithm was proposed which, allegedly, is even faster than insertion sort [Knowles et al. 2014]; partly due to backwards memory allocation [Knowles et al. 2013].

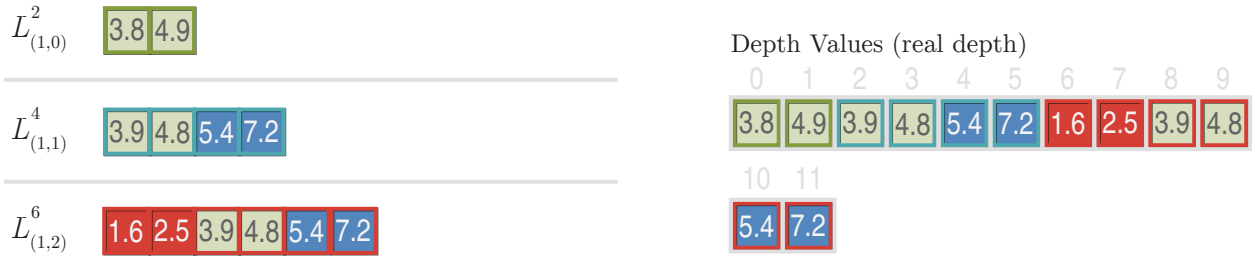
A Note on Convergence It is interesting to note how previous ideas have converged into this unified approach. The A-buffer [Carpenter 1984] lays the foundation by defining layered depth maps. The R-buffer [Wittenbrink 2001] and F-buffer [Mark and Proudfoot 2001] suggest the use of a unique storage each fragment. Lastly, the many k-buffer implementations [Callahan et al. 2005; Bavoil et al. 2007; Myers and Bavoil 2007] highlighted the need for RMW buffers and atomic operations in modern graphic pipelines.

Paging Storing multiple depth values per node can decrease the memory overhead of the singly linked list approach [Crassin 2010b]. These so-called *paged per-pixel singly linked lists* (PPPSLL) require fewer nodes overall thus decreasing the memory used by the integer indexing. However, a new auxiliary buffer must be introduced to store the number of depth values per pixel so that the algorithm will know when to allocate a new list node.

The paged approach may perform better than the non-paged alternative. It depends on the scene's depth complexity and the page size. We will not go into further detail with paging but simply mention the approach here for completeness.

[5] C-like pointer indirection is available through non-standard Nvidia extensions [Bolz et al. 2010; Brown 2012].

[6] This does not cause any ambiguities since the 0 index is reserved for a head index. Thus a list node can't refer to index 0 as if it was another node.



(a) *Using the example seen in Figure 3. Each L_p^k sequence is stored in a contiguous variable-length array.*

(b) *Memory layout. Using the example seen in Figure 9a. The L_p^k sequences are stored back-to-back. Not pictured is the auxiliary structure which maps the fragments to pixels.*

Figure 9: *l-buffer.*

Depth Ranges Using a single buffer to store all list nodes conserves memory. However, the high frequency of access to said buffer may cause contention. The solution is to division the scene into depth ranges and have a buffer for each range [Vasilakis and Fudos 2013]. Thus contention is reduced as a function of the number of depth ranges. On the other hand, more memory may be wasted due to unfilled buffers (and non-uniform depth distribution).

As with paging, the optimal number of divisions depend on the scene's depth complexity. We will not go into further details with depth ranges and merely mention it for completeness.

3.2.9 l-buffer

The *layered buffer or list buffer* (l-buffer) is a pointerless A-buffer derivative [Lipowski 2010]. Like the A-buffer, the l-buffer implements L_p sequences (without a fixed k). Thus the l-buffer can store an unlimited amount of fragments per pixel. Depth values are stored in per-pixel contiguous variable-length arrays thus not requiring any pointer indirection (similar to Z^3). Unlike Z^3 (Figure 6a), however, the length of said arrays match the actual depth complexity (Figure 9a).

Construction A direct comparison between the A-buffer (Figure 5), Z^3 (Figure 6a), and the A-buffer (Figure 9a) would seem to strongly favor the l-buffer. The latter is seemingly the best of both worlds: The unfixed L_p sequence from the A-buffer and the contiguous, pointer-less layout of Z^3. The downside is the complicated construction.

The proposed l-buffer construction requires at least seven passes [Lipowski 2010]. Contrast this to the PPSLL's two-pass construction. However, the l-buffer targets OpenGL 3.x hardware which is the main reason for the complex construction. Specifically, the proposed l-buffer construction does not make use of either SSBOs or atomic operations. The DF-buffer will show that OpenGL 4.x hardware can shave off three passes. As such, the proposed l-buffer construction is archaic. Still, it is remarkable that the l-buffer can actually be constructed without RMW. We will give a cursory outline of the proposed passes for comparison with newer methods:

1. **Fragment Counting.** The scene geometry is rasterized into a *count buffer*. The latter is a per-pixel fragment count. The count buffer is implemented via additive stencil operations.

2. **Reduction.** The count buffer is reduced into a *maximum per-pixel fragment count*, `max_count`. This is done through a series of recursive max-operations done in parallel.

3. **Buffer Initialization.** A *layered buffer* is initialized. The layered buffer stores L_p^k sequences with $k =$ `max_count`. The memory layout for the layered buffer is essentially that of Z^3.

4. **Fragment Storing.** The scene geometry is rasterized into the layered buffer. An auxiliary stencil buffer is used to route the incoming fragments to consecutive layers. Note that the *layered buffer* may contain many null values (Figure 6a).

5. **Prefix Sum.** The null values must be skipped. To do so, a prefix sum over the *non-empty* pixels in the layered buffer is used to produce per-pixel offsets denoting the location of the consecutive fragments. This requires the scene geometry to be rasterized. The offsets are stored in a *map buffer*.

6. **Condensation.** The map buffer is used to index into the layered buffer in order to produce a dense, one-dimensional list of fragments. The latter is called the *list buffer* which is the final product. All other intermediary buffers are discarded.

7. **Sorting.** The fragments in the list buffer must be depth-sorted in order to conform to the L_p definition.

Note that steps 1–3 result in the construction of a Z^3-like buffer (it's only missing the two derivative z-values). This is interesting, since Z^3 was originally proposed as a hardware extension [Jouppi and Chang 1999]. Unfortunately, this also implies that a fixed k is used for the layered buffer. Thus steps 1–6 must be repeated if overflow is detected.

Memory The l-buffer's final memory layout is the densest yet (Figure 9b). This picture is not entirely true, however, since some additional data is needed to map each depth value to its corresponding pixel. There are two solutions:

- Construct an offset buffer (similar to the map buffer) and use it to index into the l-buffer (Figure 11). The required memory is a function of the viewport size.

- Store the fragments xy-coordinates directly in the l-buffer. The required memory is a function of the fragment count.

Both solutions require additional memory. Which approach to choose depends on the scene's complexity.

Even though the final memory requirements are small, the intermediary requirements are huge due to all the auxiliary buffers. Especially the **Buffer Initialization** step which requires the construction of a Z^3-like buffer.

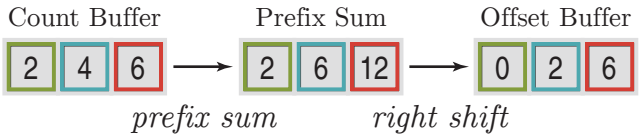


Figure 10: *Computation of the offset buffer. Using the example seen in Figure 3. A prefix sum followed by a right shift with zero-saturation. Note that the right shift can be implemented as part of the prefix sum operation.*

Sorting The l-buffer authors do not suggest any sorting algorithm. Any algorithm can be applied. E.g., a local insertion sort as used with PPSLL.

A Note on Novelty The idea to linearize the memory layout using a prefix sum first appeared as a hardware extension in a patent application [Peeper 2008]. The l-buffer authors do not cite the patent application so we can only assume that they developed their method independently.

3.2.10 DF-buffer

The *dynamic fragment buffer* (DF-buffer) is essentially an l-buffer with a more efficient construction method that utilizes OpenGL 4.x hardware [Comba et al. 2012]. The DF-buffer is constructed in four steps (in contrast to the l-buffer's proposed seven steps). We outline the steps below:

1. **Fragment Counting**. The scene geometry is rasterized. A *count buffer* stores per-pixel atomic counters, `count`, which count the number of fragments per pixel.

2. **Prefix Sum.** The CUDA-accelerated Thrust library is used to compute the prefix sum of the count buffer in parallel. The resulting offsets, `offset`, are stored in an *offset buffer* (Figure 10).

3. **Fragment Storing.** The scene geometry is rasterized. The incoming fragments are stored in the DF-buffer. The memory location can be determined as `offset + (--count)`. Here, `offset` and `count` are found by a lookup into the offset buffer and count buffer, respectively. Note that the decrement of `count` must be atomic.

4. **Sorting.** Insertion sort is used to sort the fragments.

The **Reduction** step is made redundant by atomic counters. The **Buffer Initialization** step has also become redundant since the layer buffer is not needed. Furthermore, the count buffer is automatically reset to zero due to the decrements in step three. The DF-buffer is constructed directly in step three, so no **Condensation** step is required. Note that the offset buffer is kept so that it can be used to map each pixel to its corresponding list of fragments (Figure 11).

The l-buffer's proposed prefix sum algorithm is actually serial (even though it's executed on the GPU). The DF-buffer uses a more efficient parallel prefix sum instead.

Even with the above-mentioned construction optimizations, the DF-buffer still requires two rasterization passes of the scene geometry (the l-buffer requires at least three geometry passes). Contrast this to the PPFLA and PPSLL which only require a single geometry pass. Thus the latter methods may be preferable for complex geometry. However, the DF-buffer is still the most memory efficient of the three.

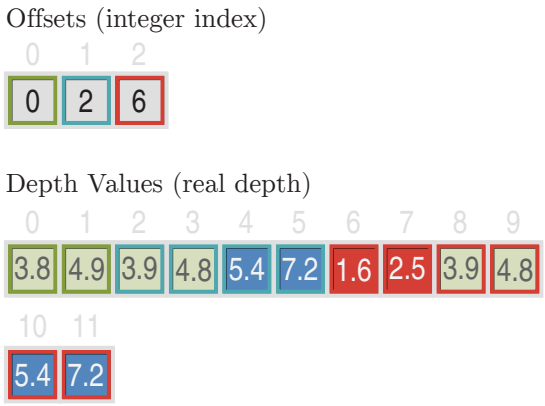


Figure 11: *DF-buffer memory layout. Using the example seen in Figure 3. Note the use of the offset buffer from Figure 10.*

3.2.11 S-buffer

The *sparsity-aware buffer* (S-buffer) is yet another l-buffer with construction optimizations [Vasilakis and Fudos 2012]. The S-buffer utilizes the same optimizations as the DF-buffer. In fact, the DF-buffer authors actually cites the S-buffer as an inspiration. The difference between the two is that the S-buffer uses another prefix-sum implementation optimized for pixel sparsity (when many pixels are without fragment).

Recall that the l-buffer computes the prefix sum serially by rendering the scene geometry. By rendering the scene geometry, only non-empty pixels are affected. The prefix sum itself is calculated by a clever use of stencil operations. The details are involved and out of scope of this report.

Similarly, the S-buffer renders the scene geometry to avoid empty pixels. Instead of stencil operations, however, the S-buffer proposes to use an atomic counter. With one such counter, the prefix sum is still serial. The idea is therefore to group the pixels and provide an atomic counter for each group. This allows the prefix sum to be calculated independently within each group (linearly within a group but in parallel between groups). Then, a prefix sum is run on the counters and the result is added to the pixels within the corresponding group (so that each pixel now contains the global prefix sum).

The combination of skipping empty pixels while still computing the prefix sum in parallel allows the S-buffer to outperform the DF-buffer [Vasilakis and Fudos 2012]. However, the optimal number of pixel groups must be found empirically for each scene. Too many groups lead to management and space overhead. Too few groups lead to mediocre parallelism. Note that when a single group is used, the prefix sum is completely serial (as it is the case with the l-buffer).

3.2.12 D-buffer

The *dequeue buffer* (D-buffer) is the successor to the l-buffer [Lipowski 2013] (by the same authors). Structurally, the D-buffer is completely identical to the l-buffer. Again, the difference is the construction method. The D-buffer authors propose three different construction methods; each targeted at different specifications (roughly):

- OpenGL 3.0

- OpenGL 3.3
- OpenGL 4.2

The OpenGL 3.0 approach is identical to the l-buffer. Likewise, the OpenGL 4.2 approach is identical to the DF-buffer. Lastly, the OpenGL 3.3 approach is an intermediary hybrid of the l-buffer and DF-buffer.

As such, the main contribution of the D-buffer is the in-depth technical explanation of the three construction methods along with various micro-optimizations of said methods. Therefore, we won't go into further detail with the D-buffer.

3.2.13 HA-buffer

The *hashed A-buffer* (HA-buffer) is a hash map of depth values [Lefebvre et al. 2013]. No depth restrictions are imposed, so the HA-buffer stores L_p sequences (with unfixed k).

Specifically, the HA-buffer is a coherent, spatial hash map. Spatial, because each fragment's xy-coordinate is used as the hash key. Coherent, in the sense that neighbouring keys map to neighbouring values. In combination, neighbouring pixels will store data in neighbouring memory locations (thus exploiting locality of reference).

Construction Like PPSLL, the HA-buffer only requires a single pass over the scene geometry. The hash table itself, H, is actually a simple contiguous array of entries stored in an SSBO. It is the operations (described next) on H that defines the hash table.

Let $h(p, a)$ be the hash function where p is the hash key (the fragment's linearized xy-coordinates) and a is the entry's so-called *age*. The latter is used to resolve collisions (when multiple fragments map to the same hash value). The algorithm proceeds as follows:

1. **Clear.** All entries in H is set to zero.
2. **Fragment Storing.** Each incoming fragment is hashed, $h(p, a)$, with a initially being zero. Next, insertion into H is attempted:
   - **No collision (existing entry is zero).** Simply store the age, a, and the fragment's data (e.g., the depth value) in H at memory location $h(p, a)$.
   - **Collision (existing entry is non-zero).** If the existing entry is younger (smaller a) then mark it and take its place. Otherwise (larger a), mark the incoming entry. The marked entry ages ($a = a + 1$) and is reinserted at the next $h(p, a)$.

The aging scheme is based on the so-called Robin Hood strategy [Lefebvre et al. 2013]. By always evicting younger entries, the maximum age across the hash table is minimized. In turn, fewer reinsertions are required. The age test and eviction can be done simultaneously with a single `atomicMax` operation. This is both efficient and free of race conditions.

Because the hash map is spatial, it is easy to find all the fragments belonging to pixel, p, through iteration. That is, to compute $h(p, a)$ for each age, $a = [0; A]$, where A is a predefined max age. Thus the HA-buffer doesn't require additional buffers for indirection. Said indirection is built-in.

Coherence is achieved by the choice of h,

$$h(p, a) = p + o_a \mod |H|$$

where $|H|$ is the size of the hash table (the size of the contiguous array). o_a is a predefined set of random offsets. Note that o_a does not depend on p. Thus neighbouring keys will also have neighbouring entries in H. E.g., the neighbouring keys 0 and 1 will map to the neighbouring entries $h(0, a) = o_a$ and $h(1, a) = o_a + 1$.

Isolating p leads to an important insight

$$p = h(p, a) - o_a \mod |H|$$

Namely, that the entry's location and age ($h(p, a)$ and a) uniquely identifies which pixel it originated from. The authors dub this the *age equivalence property*. Note that $|H| \geq |V|$ must be true for this to hold, where$|V|$ is the viewport size (e.g., 800×640). Otherwise, the pixel coordinates will overlap in H. By exploiting the age equivalence property, the hash key, p, doesn't need to be stored in H (since p can be derived from $h(p, a)$ and a). This conserves memory.

Memory The memory requirements are very flexible. The only invariant is that $|H| \geq |V|$ (for the age equivalence property) which puts a lower bound on storage size. In practice, however, it must be that $|H| \gg |V|$ since multiple fragments can map to each pixel. The optimal value of $|H|$ must be found empirically. A large $|H|$ will use up a lot of memory but reduce the number of hash collisions and thus increasing performance. Analogously, a small $|H|$ conserves memory but results in many collisions.

Unfortunately, there is no upper bound on $|H|$. Naturally, the upper bound should be the total number of fragments but that metric is not available during rasterization.

Sorting The observant reader may have noticed that the **Sorting** step is missing. As it turns out, this step can be skipped due to the age equivalence property. Recall that the hash key doesn't have to be stored in H due to the age equivalence property. Thus the entries of H will be tuples of a, depth value, and data. By enforcing that exact order (compressed into a `uint32_t` or `uint64_t`), the above-mentioned hash map is automatically depth-sorted during construction. This is because the `atomicMax` operation will first compare age (in the most significant bits) and then depth (in the subsequent bits). In other words, the collision resolution step is overloaded to also do depth-sorting (without any additional overhead). This algorithm is essentially an insertion sort (the same algorithm used to construct PSPPFLA).

The downside is that a (8 bits) and the depth value (24 bits) can just fit into a `uint32_t`, leaving no room for additional data (e.g., fragment color). Using a `uint64_t` leaves 32 bit spare for data storage. As mentioned previously, however, the 64-bit `atomicMax` operation is a new addition to consumer hardware and not yet wildly adopted [Liu et al. 2010; Lefebvre et al. 2013].

3.2.14 Pre-sorted Per-pixel Singly Linked Lists

Insertion sort can also be during construction of PPSLL to get *pre-sorted per-pixel singly linked lists* (PSPPSLL) [Lefebvre et al. 2014]. Note that this is not referring to the earlier mention of a post-process insertion sort. Please refer to Section 3.4.2 for the details.

3.2.15 Pre-sorted Per-pixel Variable-length Arrays

Per-pixel variable-length arrays can be pre-sorted (e.g., l-buffers, DF-buffers, S-buffers, and depth buffers) in order to construct *pre-sorted per-pixel variable-length arrays* (PSPPVLA) [Kubisch 2014]. The authors leave the implementation as an exercise for the user. Theoretically, one could just apply the same pre-sort approach used in PSPPFLA (see Section 3.2.7).

3.2.16 Further Hardware Advancements

Intel's pixel synchronization extension [Grajewski et al. 2013] provides efficient general-purpose critical sections in fragment shaders; exactly what is needed for parallel construction of data structures [Salvi 2013]. Unfortunately, Intel's pixel synchronization extension is currently only available on the latest incarnations of the Intel Iris and AMD Radeon GPUs [Riccio 2015]. It should be noted that Nvidia has proposed a similar extension though with slightly different syntax and semantics [Brown et al. 2014b]. The Nvidia extension, however, is only available for Nvidia's Maxwell GPUs [Riccio 2015].

Alternatively, one can use per-pixel spin locks to synchronize the fragment shaders [Vasilakis and Fudos 2014; Kubisch 2014]. Spin locks, however, are detrimental to performance. One remedy is the OpenGL thread group extension [Breton et al. 2014] which can reduce lock contention by filtering out so-called helper threads [Kubisch 2014]. Unfortunately, said extension is Nvidia-only [Riccio 2015].

Thus there is no performant cross-vendor solution for critical sections in fragment shaders. This is why we still see a prevalent use of low-level atomic operations such as `atomicAdd`, `atomicMax`, etc.

3.2.17 k^+-buffer (k-buffer revisited)

The k^+-buffer is a k-buffer which utilizes the above-mentioned critical sections to avoid the undefined behaviour of RMW [Vasilakis and Fudos 2014]. As stated earlier, fixing k implies poor scene representation which may be reasonable for OIT but not for our use case. We mention the k^+-buffer for completeness and because it is the first OIT-related implementation to use general-purpose critical sections in a fragment shader.

3.2.18 OIT-specific Methods

Some techniques and optimizations are only applicable to OIT and not to the construction of layered depth maps. We list them here for completeness.

Depth Peeling Depth peeling is a multi-pass technique which iterates over each layer (each k in L_p^k). That is, it "peels" the scene apart layer for layer from front to back (in terms of depth). Each layer is processed in isolation and **over** can be directly applied to produce OIT [Mammen 1989; Everitt 2001][7].

Depth peeling has low memory requirements since only a single layer needs to be in memory at a time. However, it requires k passes over the full scene geometry which is computationally expensive. Depth peeling is OIT-specific since it doesn't store the L_p^k sequences in an auxiliary data structure (depth values are discarded after each pass). Extensions exist that peels 2 layers [Bavoil and Myers 2011], 8 layers [Liu et al. 2006] and 32 layers [Liu et al. 2009b] at a time for improved performance .

Fragment-Parallel Composite and Filter Expanding the **over** operator recursively leads to an insight: OIT can be implemented as a parallel multiplicative scan followed by a parallel additive reduction [Patney et al. 2010]. This decomposition increases the parallelism which in turn increases performance.

[7] Depth peeling was invented by [Mammen 1989] along with an implementation for the Stellar Graphics GS1000. The term was later coined by [Everitt 2001] who provided an implementation for OpenGL 1.x hardware.

Alternative Blend Operators Various authors have proposed a commutative alternative to the **over** operator [Meshkin 2007; Bavoil and Myers 2011; Salvi and Vaidyanathan 2014]. When the blend operator is commutative, no sorting is needed (and thus no L_p^k sequences). This produces incorrect yet visually convincing results.

A hybrid approach shades the nearest fragments using an L_p^k sequence and the farther fragments using a commutative blend operator [Maule et al. 2013].

Stochastic Transparency The alpha value is interpreted as the probability that the fragment contributes to the pixel's final color [Enderton et al. 2011]. Thus no sorting is needed which increases performance. The downside is that the resulting image contains noise due to the stochastic sampling.

Adaptive Transparency This method maintains L_p^k sequences with a fixed k. The novelty is the overflow handling (when the number of fragments exceed k). If overflow occurs, the least significant fragment is merged with the incoming fragment [Salvi et al. 2011]. A fragment's significance is a function of the fragment's depth and alpha value. This strategy is similar to (yet far more advanced) the Z^3 approach. Like the k^+-buffer, adaptive transparency can also be accelerated by utilizing the pixel synchronization extension [Davies 2014].

Survey A slightly outdated OIT survey provides a qualitative comparison between most of the early methods [Maule et al. 2011].

3.2.19 Overview

A summary overview of the different layered depth map implementations can be found in Table 1. The target column indicates which platform or API that is targeted in the reference. Note that the implementation is not necessary limited to that target. E.g., PSPPFLA which is presented as a CUDA implementation can also be implemented in OpenGL 4.x.

Name	Reference	Geometry Passes	Memory	Sorting	Fragments Per Pixel	Empiric Parameters	Target
A-buffer	[Carpenter 1984]	1	Unbounded	Pre	Unlimited		CPU Early GPUs
Z^3	[Jouppi and Chang 1999]	1	Bounded	Pre	Limited	k	Hardware proposal
R-buffer	[Wittenbrink 2001]	1	Unbounded	Pre	Unlimited		Hardware proposal
F-buffer	[Mark and Proudfoot 2001]	1	Unbounded	Pre	Unlimited		Hardware proposal
k-buffer	[Callahan et al. 2005] [Myers and Bavoil 2007]	1	Bounded	Pre	Limited	k	OpenGL 2.x–3.x
PPFLA	[Liu et al. 2009a] [Liu et al. 2010] [Crassin 2010a]	1	Bounded	Post	Limited	k	CUDA OpenGL 4.x
PSPPFLA	[Liu et al. 2009a] [Liu et al. 2010]	1	Bounded	Pre	Limited	k	CUDA
PPSLL	[Yang et al. 2010] [Yang and McKee 2010] [Gruen and Thibieroz 2010] [Thibieroz 2011]	1	Unbounded	Post	Unlimited		CUDA OpenGL 4.x DirectX 11
PPPSLL	[Crassin 2010b]	1	Unbounded	Post	Unlimited	Page size	OpenGL 4.x
l-buffer	[Lipowski 2010]	3	Bounded	Post	Unlimited		OpenGL 3.x
DF-buffer	[Comba et al. 2012]	2	Bounded	Post	Unlimited		OpenGL 4.x
S-buffer	[Vasilakis and Fudos 2012]	2	Bounded	Post	Unlimited	Number of groups	CUDA OpenGL 4.x
D-buffer	[Lipowski 2013]	2	Bounded	Post	Unlimited		OpenGL 2.0–4.2
HA-buffer	[Lefebvre et al. 2013]	1	Unbounded	Pre	Unlimited	$\lvert H \rvert$	OpenGL 4.x
PSPPSLL	[Lefebvre et al. 2014]	1	Unbounded	Pre	Unlimited		OpenGL 4.x
PSPPVLA	[Kubisch 2014]	2	Bounded	Pre	Unlimited		OpenGL 4.x
k^+-buffer	[Vasilakis and Fudos 2014]	1	Bounded	Pre	Limited	k	OpenGL 4.x

Table 1: *Overview of layered depth map implementations.*

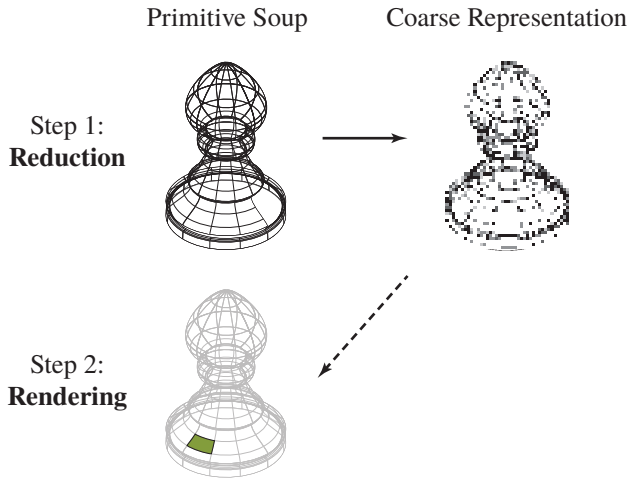


Figure 12: *Coarse scene representation in auxiliary data structure. First, the primitive soup (wireframe) is reduced (solid arrow) into a coarse scene representation (pixelated object) stored in an auxiliary data structure. Second, each primitive is rasterized in isolation (green primitive) but can query (dashed arrow) the auxiliary data structure.*

3.3 Design

Section 3 focused on layered depth maps in the context of OIT. In this section, we will analyze the use of layered depth maps as a scene representation. Additionally, we will choose the layered depth map implementation which best fits our requirements.

First, we show how a data structure of layered depth maps can be used as a scene representation. Second, we list the requirements of said data structure. Third, we go through all the layered depth map implementations and find the best match.

3.3.1 Layered Depth Maps as a Scene Representation

The complete scene representation is given by the primitive soup which is sent through the rasterization pipeline. As such, this primitive soup is the complete information available. Unfortunately, that complete information is not available at the time of fragment shading which is why local illumination models are normally used (Figure 2a).

The problem is that each primitive is processed in isolation. Thus a fragment can at best get information about its invoking primitive but not other primitives. The solution is conceptually simple (Figure 12):

1. **Reduction.** The primitive soup is reduced into a coarse scene representation and stored in an auxiliary data structure.
2. **Rendering.** The primitive soup is rasterized as normal but can now query the auxiliary data structure for global information.

Reduction is necessary since constructing a data structure with complete information would defeat the purpose of rasterization. Specifically the low memory requirements of primitive-isolated rendering. If complete information is available, ray-tracing is a better alternative. The extent of the reduction is specific to each auxiliary data structure.

Section 2 gave an overview of various scene representations and the corresponding data structures. The layered depth map is one such data structure. It can be interpreted in two ways which are presented next.

Point Cloud Interpretation For each view ray sent through the image plane, the layered depth map stores all geometric intersections in the L_p sequences (Figure 4b). Practically, only the depth values are stored but the *world coordinates* (WC) can readily be recovered [Mittring 2007]. As such, the layered depth map can be interpreted as a three-dimensional point cloud representation of the scene. The point cloud interpretation is useful for visualization (see Section 3.4.5) but the directional information is lost in the process.

Ray Set Interpretation A more practical interpretation of the layered depth map is as a set of rays and their intersections with the scene geometry. Each pixel on the image plane (each L_p sequence) corresponds to a ray. Thus the resolution of the layered depth map controls the number of rays. Similarly, the ray directions are determined by the view and projection matrices used to construct the layered depth map. That is, the rays will point in the general direction given by the forward vector (the view direction) of the view matrix. The deviation from said vector depends on the projection matrix.

With perspective projection, the rays deviate from the view direction as a function of the field of view. With orthographic projection, all the rays will be parallel with the view direction (Figure 4b). Furthermore, orthographic projection gives a uniform ray distribution. This property ensures that geometry is sampled uniformly and not as a function of depth. Contrast this to perspective projection which has a denser ray distribution closer to the camera. Moreover, orthographic projection is unbiased with respect to precision since it produces linear depth values. Consequently, orthographic projection is the better choice for a uniform scene representation in terms of both sample distribution and precision.

Multi-view The downside to orthographic projection is that all rays are parallel to the view direction. As such, each L_p sequence only provides intersection information for the view direction. Global illumination models are usually defined as an integral over a hemisphere. That is, global illumination requires samples in various directions not just one. One solution is to simply construct multiple layered depth maps each oriented in a different (Figure 13). Another solution is to ray-march through the layered depth map whose orientation is closest to the sample direction [Niessner et al. 2010].

We proceed with the multi-view approach. Note that this implies that the sample directions will be pre-defined (one for each layered depth map). This presents a challenge in that many layered depth maps must be constructed to sufficiently cover the hemisphere. We defer this discussion to Section 4.3.2.

Resolution The resolution of the layered depth map controls the number of generated rays. Thus the resolution should be large enough to adequately cover the whole scene. Of course, the resolution also has an impact on performance. Since we are using a multi-view approach, many layered depth maps will be constructed. As such, the resolution must be weighed against the number of view directions to find an optimal ratio. Moreover, we already know that the implementation will require at least one geometry pass per layered depth map (Table 1). Multiple such passes are best done in low resolution to amortize the performance cost of rasterizing the scene geometry. E.g., say the final image is 800×800 and we use 64 layered depth maps each oriented in its own direction. Then each layered depth map should be 100×100 for the total number of ras-

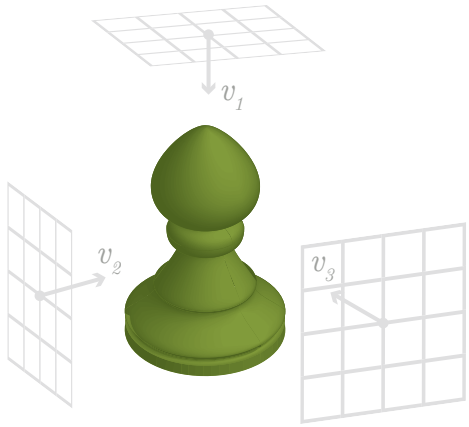


Figure 13: *Multiple layered depth maps each rendered from a different view. Here, three layered depth maps are shown corresponding to three viewing directions (v_1, v_2, and v_3). Each pixel on the image planes (4×4 in this case) will correspond to a ray. In practice, more layered depth maps of higher resolution are needed to sufficiently cover the scene.*

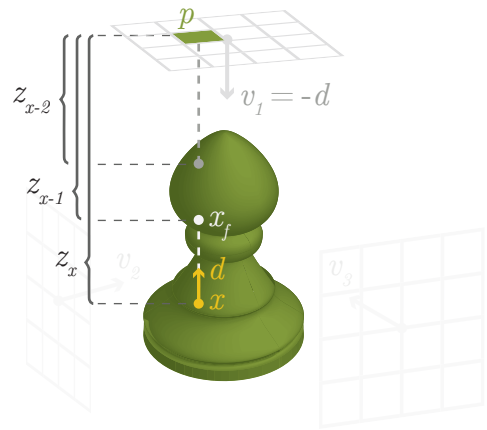


Figure 14: *Querying the multi-view layered depth maps along the ray from x in direction d. First, the layered depth map corresponding to d is found ($v_1 = -d$). Second, x is projected into the layered depth map's view to find p. Third, the $L_p = (\dots, z_{x-2}, z_{x-1}, z_x, \dots)$ sequence is searched to find z_x. Fourth, the first intersection along d is at depth z_{x-1} from which x_f can readily be constructed.*

terized pixels to match the final image

$$100 \times 100 \times 64 = 640000 = 800 \times 800$$

Of course, the complexity of the **Reduction** step may differ significantly from the **Rendering** step. Moreover, it is not a requirement that the pixel count matches; it is merely a heuristic to get decent performance. In practice, the optimal ratio of resolution weighed against the number of views is best found empirically. We defer this empirical analysis to Section 6.

Querying The main query will be ray-scene intersection tests (as done in ray-tracing). That is, given a ray, $r = (x, d)$ with initial point x and direction d, a trace should return the position of the first geometric intersection, x_f, between r and the scene geometry,

$$x_f = \text{trace}(r) = \text{trace}(x, d)$$

The trace(r) query can be broken down into four steps:

1. **Find Map.** Find the layered depth map corresponding to d.
2. **Find Pixel.** Find the pixel, p, which corresponds to x in the layered depth map.
3. **Find Depth Value.** Find the depth value, z_x, corresponding to x in the L_p sequence.
4. **Compute Intersection.** Use z_{x-1} to construct x_f.

With orthographic multi-view layered depth maps, there is direct mapping between the sample directions and the layered depth maps. As such, the **Find Map** step is trivial. The **Find Pixel** step must find the right p to sample from. The solution is to use re-projection (as done in shadow mapping). That is, x is projected from WC into NDC by the layered depth map's view-projection matrix. The NDC directly gives the position, p. In the **Find Depth Value** step, the $L_p = (z_0, z_1, z_2, \dots)$ sequence is searched to find the depth value, z_x, which corresponds to x. The key to the **Compute Intersection** step is to note that the previous depth value, z_{x-1}, belongs to the first intersection with the scene geometry. From z_{x-1} and the layered depth map's orientation, x_f can be reconstructed (Figure 14). Please refer to the Section 3.4 for further details.

Note that a single layered depth map oriented in direction v can actually be queried both in the v and $-v$ direction. The small extra step is to also find z_{x+1} which will correspond to x_f in the $-v$ direction. Algorithmically, only a single additional step has to be added. Thus both directions can be tested simultaneously with practically no overhead.

Please refer to Section 4.4.1 for an implementation of trace(r).

Summary Our auxiliary data structure consists of multiple low-resolution layered depth maps each rendered from a different view direction. This design will lay the foundation for the rest of the report. In the next section, we will set up some additional requirements that will help us in determining the right implementation for the design.

3.3.2 Requirements

The overall idea is to use layered depth maps as an auxiliary data structure that can be queried for global information during rasterization (Figure 2c). Especially, we are interested in overcoming the local limitation of rasterization. This leads us to the first requirement:

Requirement 1 *The data structure must be total; include the whole scene.*

Total, in the sense that the data structure doesn't exclude geometry outside some bounds. Not that it should capture as much geometric detail as possible (which we refer to as completeness). Of course, we would prefer that the data structure also did the latter but it's not a necessity. In fact, geometric detail should be put up against performance to find an optimal ratio. This leads us to the next requirement:

Requirement 2 *Queries on the data structure must be fast; perform in real-time.*

Being real-time allows the data structure to compete with local illumination methods. To compete with real-time methods, the data structure must also be able to adapt to scene changes. That is:

Requirement 3 *The data structure must be dynamic; adapt to geometric changes.*

Otherwise, the global illumination might as well be computed offline and baked into textures.

Secondary Requirements As stated in the introduction, we will use OpenGL 4.x for hardware acceleration. That is, we don't strive to be backward compatible with earlier API versions. Consequently, our data structure will require fairly recent hardware. We do, however, strive to deliver a cross-vender solution. I.e., we won't limit our implementation to specific hardware by using exotic OpenGL extensions.

3.3.3 Candidate Evaluation

Before even assessing the requirements, we can eliminate the early methods. Namely those that target CPUs or early GPUs (the A-buffer) and the hardware proposals (Z^3, the R-buffer, and the F-buffer). The aforementioned implementations are simply not a practical match for modern GPUs and APIs. Said implementations merely serve as historical context in our comparative survey.

Requirement 1 Analysis Requirement 1 immediately eliminates many candidate implementations: Those that limit the number of fragments per pixel. Such limitations put an arbitrary bound on the depth complexity which is in clear violation of said requirement. This further eliminates the k-buffer, PPFLA, PSPPFLA, and the k^+-buffer.

Requirement 2 Analysis Requirements 2 is about performance and is best evaluated quantitatively by profiling each implementation. Profiling, however, requires implementing all the candidates which is out of scope of this report. Instead, we use the memory layout as an approximate metric. Specifically, the amount of indirection required to find a depth value for a pixel.

PPSLL and PSPPSLL store depth values in arbitrary memory locations with each depth value pointing to the next in the list (Figure 8). Furthermore, list traversal must be done linearly so finding a depth value is an $O(n)$ operation in the worst case (regardless of sorting). This, combined with the random memory access pattern (causing cache misses) would indicate that singly linked lists are not ideal for querying depth values.

The l-buffer, DF-buffer, S-buffer, depth buffer, and PSPPVLA all store depth values in contiguous arrays. Furthermore, finding a depth value in a sorted contiguous array can be done with binary search which is an $O(\log n)$ operation in the worst case. This, combined with contiguous storage (exploiting locality of reference) would indicate that sorted contiguous arrays are the best option for querying depth values.

Lastly, the HA-buffer stores depth values in a spatial, coherent hash map. Finding a depth value requires traversing all the entries for a pixel which is an $O(n)$ operation in the worst case. Seemingly, the memory access pattern is random but the coherence between neighbouring pixels helps to keep the cache warm. Combined, this indicates that a hash map of depth values is a fitting structure for querying depth values. Thus the hash map lies somewhere in between singly linked lists and sorted contiguous arrays in terms of querying performance.

Surprisingly, singly linked lists are often found to perform on par with variable-length contiguous arrays [Knowles et al. 2012; Vasilakis and Fudos 2014; Kubisch 2014]. Specifically, [Knowles et al. 2012] reports a performance difference of only 10 % in favor of the variable-length contiguous arrays. Though much larger differences have also been reported [Vasilakis and Fudos 2012]. Note that these reports are based on unsorted arrays used for OIT purposes. As such, the results can't be directly applied to our use case. However, one conclusion still holds: The memory access pattern of singly linked lists is not detrimental to performance in practice. One explanation is that the allocation of list nodes is grouped by the threads working on the same primitive [Knowles et al. 2014]. Thus the list nodes are actually coalesced in practice and locality of reference can be exploited.

Requirement 3 Analysis All the considered implementations produce static structures of layered depth maps. Indeed some implementations allow depth values to be inserted after construction but no implementation supports an efficient remove operation. Consequently, all of the implementations seem to violate Requirement 3. However, if the layered depth maps are constructed each frame then the implementation complies with Requirement 3. Thus Requirement 3 should actually be evaluated by the efficiency of each implementations' construction method. As stated earlier, performance is best evaluated quantitatively by profiling but this is an immense task. Instead, we use the number of geometry passes and the sorting requirements as an approximate metric.

The fewer geometry passes, the better performance. With this in mind, PPSLL, PPPSLL, the HA-buffer, and PSPPSLL are the top candidates. This is, of course, a rough heuristic since the complexity of each pass is left out. Still, a geometry pass requires full scene rasterization which is a significant overhead. We assume that the combined cost of two such passes will vastly outweigh even the most complex linked list or hash map implementation.

The sorting requirements are more difficult to assess. Post-sorting requires that all fragments are copied to a shader-local array which is subsequently sorted. Said array must be of fixed length, say K, since dynamic allocation in shaders (through SSBOs) is expensive. If there are more than K fragments, then sorting can be done in place (in SSBO memory). However, this is detrimental to performance [Thibieroz 2011; Knowles et al. 2012]. Even if there are less than K fragments, the sorting step has been observed to be the main bottleneck in the layered depth map construction [Knowles et al. 2012; Knowles et al. 2014]. Note that the use of a fixed K also violates Requirement 2. Pre-sorting does not impose such a limit. Pre-sorting, however, requires expensive atomic operations to implement. Still, we assume that the overhead of atomic operations is much lower than the cost of a post-processing step. This is also supported by empirical evidence in the case of PPSLL versus PSPPSLL [Lefebvre et al. 2014]. Lastly, the post-sorting pass can be combined with the blending pass when doing OIT. The same is not true for our use case of layered depth map. In fact, either the sorting must be done in place (which is expensive) or an extra pass is needed to put the sorted values from the shader-local array back into the layered depth map in order. Thus post-sorting is further penalized for adding the aforementioned overhead.

3.3.4 Candidate Selection

All requirements considered, we find that PSPPSLL are the best choice for our use case. PSPPSLL do not impose limits on the number of fragments per pixel (Requirement 1), are fast to query in practice (Requirement 2), and can be efficiently constructed (Requirement 3).

While the l-buffer's (and similar buffer's) sorted contiguous arrays are faster to query (through binary sort), we deemed the construction costs to be too high for our purposes. Our auxiliary data structure is based on multiple layered depth maps and for each map we pay the

construction costs. Thus the latter must be kept as low as possible which is why we favor singly linked lists.

The HA-buffer is also a prime candidate which has almost identical characteristics compared to PSPPSLL (Table 1). The only difference is that the HA-buffer relies on an additional empiric parameter: The hash map size, $|H|$. As such, we favor the PSPPSLL since it is one less scene-dependent parameter to worry about. Another key factor is that PSPPSLL reportedly performs better overall than the HA-buffer [Lefebvre et al. 2013; Lefebvre et al. 2014]. Part of this performance difference is due to driver bugs with atomic operations which require the insertion of unnecessary memory barriers. Consequently, the performance comparison isn't completely fair but definitely a practical factor to note. We too found that driver issues impose unnecessary limits in practice (Section 3.4.3).

Completely left out by Requirement 1, are the fixed-length arrays. These methods are otherwise prime candidates because of their unparalleled performance. All reports always list the fixed-length arrays as one of the top performer compared to singly linked lists and variable-length arrays [Knowles et al. 2012; Vasilakis and Fudos 2014]. If not for Requirement 1, those methods would also be good candidates. However, even if we assume that k can be chosen so that no fragments are omitted, the memory requirements would be enormous. As stated earlier, most of that memory would be wasted storing null values. With singly linked lists, exactly the right amount of memory is used (though with overhead due to indirection).

Memory Requirements Note that we have chosen a method with unbounded memory requirements. That is, a method for which we do no know the memory requirements before construction. As mentioned earlier, dynamic memory allocation on the GPU is impractical. Our solution is instead to use a memory pool, U, for the unbounded methods. Let M be the total memory of the application and let B be the memory used by all bounded allocations (e.g., meshes, textures, RTs). Then the pool is allocated to use all the remaining memory,

$$|U| = |M| - |B|$$

Thus being unbounded is not concern in practice provided the application has a memory budget (so that $|M|$ is known). The application should of course report an error of the memory pool overflows. Note that the memory pool, U, may be shared between multiple unbounded methods.

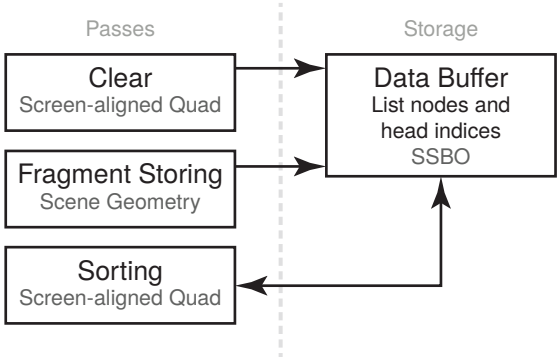


Figure 15: *PPSLL construction.*

3.4 Implementation

In this section, we will present an OpenGL 4.x implementation of PSPPSLL. First, we explain how to implement PPSLL. Second, we extend the implementation with pre-sorting. Third, we discuss driver issues and how they can be circumvented, Fourth, we list some practicalities that must be accounted for in the actual implementation. Fifth, we show how a layered depth map can be visualized as a point cloud.

3.4.1 PPSLL Construction

The concurrent construction of singly linked lists has had a GPU implementation for a long time [Harris 2001]. The method we describe is based on a simplification of [Harris 2001]. Specifically, only `push_front` operations are supported (as the layered depth map are reconstructed every frame). Our implementation is semantically equivalent to the original CUDA implementation of [Yang et al. 2010] but implemented using OpenGL 4.x.

Construction As mentioned earlier, only a single SSBO is required to store both head indices and list nodes (Figure 8). Still, we logically split head indices and list nodes into two buffers:

- The `head_buffer` contains a head index for each pixel.
- The `node_buffer` contains all list nodes (fragment data and an index to the next node).

The start of each singly linked list can be queried from the `head_buffer`. Initially, all lists are empty (the `head_buffer` is cleared to zero). The `node_buffer` is where the actual data is stored. The `head_buffer` is logically a two-dimensional array since it contains an entry for each pixel whereas the `node_buffer` is a one-dimensional data array. Note that the `node_buffer` is unbounded (as discussed earlier). For now, just assume that the `node_buffer` is large enough to contain all the list nodes. Additionally, the method requires an atomic counter, `count`, initialized to zero each frame. `count` will keep track of the memory used in the `node_buffer`.

Three passes are required [Yang et al. 2010; Yang and McKee 2010; Gruen and Thibieroz 2010; Thibieroz 2011]:

1. **Clear (screen-aligned quad).** The per-pixel head indices are set to zero (which denotes the end of the list).
   (a) A simple fragment shader sets each `head_buffer` entry to zero.
2. **Fragment Storing (scene geometry).** Each incoming fragment is stored in the singly linked list.
   (a) Allocate memory for a new list node, `new_node`, in the `node_buffer`.
   (b) Store any needed data in `new_node` (e.g., the fragment's depth value, color, etc.).
   (c) Update the head index in the `head_buffer` to point to `new_node`.
   (d) Set `new_node`'s next index to the old head index.
3. **Sorting (screen-aligned quad).** The per-pixel singly linked lists are copied into a shader-local array and subsequently sorted.

Please refer to 3.4.1 for an overview of the passes. We skip the explanation of the **Sorting** step since we will soon present a pre-sorted version (Section 3.4.2). The **Clear** pass is trivially implemented. The **Fragment Storing** pass is more involved and is explained in detail in the following paragraphs. Before that, note that the **Fragment Storing** step is actually a `push_front` operation. That is, the head index is updated in each step.

(a) The shader invocation needs a unique place to store `new_node` in the `node_buffer`. I.e., a unique integer index into the `node_buffer`. This is the purpose of the atomic counter, `count`, and can be implemented with the `atomicCounterIncrement` operation. I.e.,

```
uint32_t atomicCounterIncrement(atomic_uint c)
```

which atomically increments `c` and conveniently returns the old value. Thus all that is required is to call `new_node = atomicCounterIncrement(count)`. Now, `new_node` stores a unique integer index into the `node_buffer`. Note that this must be done atomically to ensure that the shader invocation has exclusive access to `new_node`. Otherwise, two shader invocations may get the same index and corrupt the stored data.

(b) Since `new_node` indexes into a unique chunk of memory, the shader invocation can safely store any fragment data without worrying about data races. Thus no atomic operations are needed for this step.

(c) The head index can be accessed via the shader invocation's xy-coordinates. As for the update, the helpful `atomicExchange` function is needed. I.e.,

```
uint32_t atomicExchange(inout uint32_t a, uint32_t b)
```

which atomically sets `a` to `b` and returns the old value of `a`. Thus the head index update simply becomes a call to `old_head = atomicExchange(head, next)`. Note that we save the old head index, `old_head`, for later. Again, it is crucial that this is done atomically. Otherwise, two shader invocations may attempt to set `head` at the same time which results in a race condition.

(d) Simply assign `new_node`'s next index to `old_head`. This assignment does not need to be atomic since the shader invocation has exclusive access to `new_node`. This is the same reasoning behind step (b).

Source Code Pseudo-code for the **Fragment Storing** step is given in Listing 1. Note that steps (b) and (c) have been combined. Thus `old_head` is no longer needed.

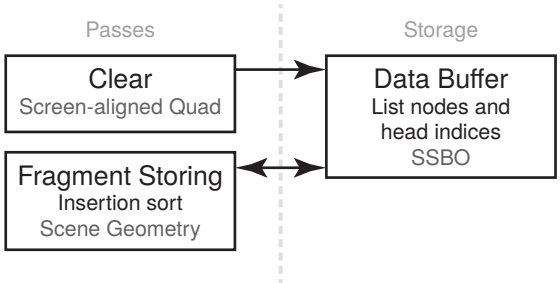


Figure 16: *PSPPSLL construction.*

GLSL code for a fragment shader implementing the **Fragment Storing** step is given in Listing 2. The `head_buffer` and `node_buffer` are merged into a single `data` buffer in practice (Figure 8). Therefore, the allocation routine must now offset the returned index to skip over the head indices. Note that the fragment shader only writes to SSBO memory and not to the framebuffer. Also note the qualifiers for the `data` buffer. The `coherent` keyword ensures that reads and writes are coherent with other shader invocations [Kessenich et al. 2014]. Practically, coherent reads and writes defeat the cache and directly accesses the underlying memory. This ensures that the shader invocations will see the updates to the head indices. The `restrict` keyword tells the compiler that the `data` variable is the only way to access the underlying storage. [Kessenich et al. 2014]. The compiler can use this information for optimizations.

3.4.2 PSPPSLL Construction

The **Fragment Storing** step in PPSLL construction can be modified to use insertion sort and thus producing PSPPSLL [Lefebvre et al. 2014]. The new step proceeds as follows (Figure 16):

2. **Fragment Storing (scene geometry).** Each incoming fragment is stored in the singly linked list.
   (a) Allocate memory for a new list node, `new_node`, in the `node_buffer`.
   (b) Store any needed data in `new_node` (e.g., the fragment's depth value, color, etc.).
   (c) Traverse the list until the stored node's depth value is larger than incoming fragment's depth value.
   (d) Insert `new_node` at this location.

Of course, the **Sorting** step can now be omitted resulting in two passes total (one geometry pass). Each sub-step is explained in detail in the following paragraphs.

(a) and (b) These two steps are exactly as before and doesn't have to be changed.

(c) The list is traversed by following each node's `next` index until either:

- The end of the list is reached.
- The stored node's depth value is larger than the incoming fragments's depth value.

In practice, an infinite loop wraps the node iteration and a `break` statement is used to end the procedure. Two variables are used to keep track of the current node (`current`) and the previous node (`previous`). We need to keep track of both in order to insert the `new_node` into the list. Initially, `previous` is the head node and `current` is the first node in the list, `head_buffer[head].next`, or zero if the list is empty. In each iteration, the above-mentioned tests are performed

```
// Step (c)
for(;;) {
    // We are either at the end of the list or just before a node of greater depth...
    if (current == 0 || depth < node_buffer[current].depth) {
        // ...so we insert the new node here.
        // Step (d)
    }

    // We are still searching for a place to insert the new node...
    else {
        // ...so we advance to the next node.
        previous = current;
        current = node_buffer[current].next;
    }
}
```

We will expand on the insertion routine in the next step.

(d) The simplest (and incorrect) approach is to store the node as one would do in a serial implementation

```
// The new node is inserted before the current node.
node_buffer[new_node].next = current;
// The previous node is updated to the new node.
node_buffer[previous].next = new_node;
```

The first line of code is correct. Recall that the `new_node` is a unique index and as such the shader invocation has exclusive access to that part of the `node_buffer`. Thus only the current shader invocation will read and write to `node_buffer[new_node].next`. The second line of code, however, has a race condition. The memory at the `previous` index is shared between multiple shader invocations. Consequently, two shader invocation may attempt to write to `node_buffer[previous].next` at the same time resulting in a race condition.

We saw before that this can be fixed with an `atomicExchange` operation. Applying this logic leads to the following (also incorrect) implementation

```
// The new node is inserted before the current node.
node_buffer[new_node].next = current;
// The previous node is updated to the new node.
atomicExchange(node_buffer[previous].next, new_node);
```

Indeed, the writes to `node_buffer[previous].next` will now always be atomic but another race condition is still present. Say that one shader invocation has found the right place to insert its fragment and is just about to write to `node_buffer[previous].next`. Then a second shader invocation catches up with the first, inserts its fragment into the list, and exits. The first shader invocation is unaware of the actions of the second shader invocation and proceeds with the `node_buffer[previous].next` operation as if nothing had happened. Unfortunately, the work of the second shader invocation has now been overwritten by the first and a node.

Now, we present the correct solution which makes use of an `atomicCompSwap` operation. I.e.,

```
uint32_t pixel_index = /* get from the fragment's xy-coordinates */
// Step (a)
uint32_t new_node = atomicCounterIncrement(count);
// Step (b)
node_buffer[new_node].data = /* depth value, color, etc. */
// Step (c) and (d)
node_buffer[new_node].next = atomicExchange(head_buffer[pixel_index], new_node);
```

Listing 1: *Pseudo-code for the **Fragment Storing** step of PPSLL construction.*

```
struct list_node {
    uint32_t next;
    /* omitted */ data;
};

layout(/* omitted */) coherent restrict buffer data_buffer {
    list_node data[];
};

layout(/* omitted */) uniform atomic_uint count;
uniform uint32_t width, height; // Viewport size

uint32_t allocate() {
    // The head indices are stored first, so the returned index
    // is offset by the viewport size.
    return width * height + atomicCounterIncrement(count);
}

void main() {
    uint32_t pixel_index = /* get from the fragment's xy-coordinates */
    // Step (a)
    uint32_t new_node = allocate();
    // Step (b)
    data[new_node].data = /* depth value, color, etc. */
    // Step (c) and (d)
    data[new_node].next = atomicExchange(data[pixel_index], new_node);

    // Note that there are no writes to the fragment buffer.
    // Only SSBO storage is used.
}
```

Listing 2: *GLSL fragment shader for the **Fragment Storing** step of PPSLL construction.*

```
uint32_t atomicCompSwap(
    inout uint32_t a,
    uint32_t b,
    uint32_t c)
```

which atomically sets a to c if a == b. Otherwise (a != b), a is unmodified. The operation always returns the previous value of a (regardless of the comparison). The correct solution is

```
// The new node is inserted before the current node.
node_buffer[new_node].next = current;

/* barrier */

// The previous node is updated to the new node.
uint32_t previous_next = atomicCompSwap(
    node_buffer[previous].next,
    current,
    new_node);

// The atomic update occurred...
if (previous_next == current)
    // ...so we are done.
    break;
// Another thread updated node_buffer[previous].next
     before us...
else
    // ...so we continue from previous_next
    current = previous_next;
```

The update on the previous is now just attempted with the conditional atomicCompSwap operation. If the update occurred, the insertion is done. If the update fails, some other shader invocation must have caught and the list traversal simply continues from the newly inserted node.

After a successful atomicCompSwap operation, the inserted node is made visible to other shader invocations. Therefore, it is important that the write to node_buffer[new_node].next occurs before the atomicCompSwap operation. Otherwise, a second shader invocation may catch up and read node_buffer[new_node].next before it has been written to by the first shader invocation. This leads to undefined behaviour. Theoretically, the compiler (or instruction pipeline) may reorder the instructions so that the write occurs after the node has been made visible. Thus for complete conformance, a memoryBarrier() should be inserted at the /* barrier */ line. In practice, however, it is not always necessary to add the memory barrier as some systems do not reorder the critical instructions anyhow. For those systems, the barrier can be omitted for added performance.

Finiteness Having an infinite loop in a shader may seem intimidating at first. The shader is, however, guaranteed to finish in a finite number of steps. This is because a shader invocation never loses progress. Progress may be stalled due to overlapping insertions (at the atomicCompSwap operation) but the stalled shader will always pick up from the newest update (the value of previous_next). As such, the insertions will at worst be executed sequentially. Fortunately, this is not the case in practice. We explore this topic further in the next paragraph.

Lock-free The algorithm finishes in a finite yet indeterminate number of steps. Indeterminate, since some steps may be repeated due to concurrent updates. Thus the total number of steps required for a single insertion is not known beforehand. However, finiteness guarantees that the algorithm terminates eventually. In other words, it is a lock-free algorithm (though not wait-free). Contrast this to algorithms which rely on critical sections (and hence locks). The fine-grained atomic operations of the **Fragment Storing** step allows for concurrent insertions which in turn increases the parallelism. A critical section around the loop, while intuitive and simple, would effectively make the **Fragment Storing** step sequential.

Source Code GLSL code for a fragment shader implementing the updated **Fragment Storing** step is given in Listing 3. Again, the node_buffer and head_buffer is merged into a single data buffer in practice.

3.4.3 Driver Issues

The GLSL code provided in Listing 3 may need further modification to work. We found that the atomic operations had race condition issues for seemingly random combinations of resolution and the number of layered depth maps being constructed. This happened on both an Nvidia GeForce GTX 480 (driver version 344.75) and GeForce GTX 780 Ti (driver version 347.25). Driver version 347.25 is the latest at the time of writing. In practice, this caused random nodes to be dropped from the layered depth map. Note that this is not due to omitting the memory barrier. The results were the same with the memory barrier left in. In fact, we tried inserting memory barriers after every statement to no avail.

The workaround is to replace a read operation with an unnecessary atomic operation. Specifically, we replaced the line

```
current = data[current].next;
```

with

```
current = atomicAdd(data[current].next, 0);
```

in the else statement inside the loop. Both lines are semantically equivalent. Furthermore, there is no race condition at that line. Still, using the atomicAdd circumvented the issues in practice. First, we suspected that the compiler must have reordered instructions and that the atomicAdd operation had forced the correct ordering. However, a quick look at the assembly refutes this explanation. The following fragment is an assembly excerpt of the else statement using a plain read

```
ELSE;
MUL.S R0.w, R0.x, {12, 0, 0, 0}.x;
MOV.U R0.y, R0.x;
MOV.U R0.x, R0.w;
LDB.U32 R0.x, sbo_buf0[R0.x];
ENDIF;
```

The following assembly is the same else statement but using the atomicAdd operation

```
ELSE;
MUL.S R0.w, R0.x, {12, 0, 0, 0}.x;
MOV.U R0.y, R0.x;
MOV.U R0.x, R0.w;
ATOMB.ADD.U32 R0.x, {0, 0, 0, 0}, sbo_buf0[R0.x];
ENDIF;
```

Note that no instructions have been re-ordered in the assembly. The only difference is that the LDB.U32 instruction has been replaced with ATOMB.ADD.U32. The remaining assembly in its entirety is completely identical. Still, it is possible that some instructions are reordered at run-time in the GPU's instruction pipeline. However, a memory barrier should prevent such reordering (and we tried adding memory barriers to no avail).

```
struct list_node {
    uint32_t next;
    /* omitted */ data;
};

layout(/* omitted */) coherent restrict buffer data_buffer {
    list_node data[];
};

layout(/* omitted */) uniform atomic_uint count;
uniform uint32_t width, height; // Viewport size

uint32_t allocate() {
    // The head indices are stored first, so the returned index
    // is offset by the viewport size.
    return width * height + atomicCounterIncrement(count);
}

void main() {
    uint32_t pixel_index = /* get from the fragment's xy-coordinates */
    // Step (a)
    uint32_t new_node = allocate();
    // Step (b)
    data[new_node].data = /* depth value, color, etc. */

    // Start with the head node
    uint32_t head_node = width * height + pixel_index;
    uint32_t previous = head_node;
    uint32_t current = data[head_node].next;

    // Step (c)
    // Insert the new node while maintaining a sorted list.
    for (;;)
        // We are either at the end of the list or just before a node of greater depth...
        if (current == 0 || depth < data[current].depth) {
            // Step (d)
            // ...so we attempt to insert the new node here. First,
            // the new node is set to point to the current node. It is crucial
            // that this change happens now since the next step makes
            // the new node visible to other threads. That is, the new node must
            // be in a complete state before becoming visible.
            data[new_node].next = current;
            // Memory barrier omitted for added performance.

            // Then the previous node is atomically updated to point to new node
            // if the previous node still points to the current node.
            // Returns the original content of data[previous].next (regardless of the
            // result of the comparison).
            uint32_t previous_next = atomicCompSwap(data[previous].next, current, new_node);

            // The atomic update occurred...
            if (previous_next == current)
                // ...so we are done.
                break;
            // Another thread updated data[previous].next before us...
            else
                // ...so we continue from previous_next
                current = previous_next;
        // We are still searching for a place to insert the new node...
        } else {
            // ...so we advance to the next node in the list.
            previous = current;
            current = data[current].next;
        }
}
```

Listing 3: *GLSL fragment shader for the* ***Fragment Storing*** *step of PSPPSLL construction.*

Consequently, we can only explain the race condition as a driver issue. Such issues with atomic operations have been reported before [Lefebvre et al. 2014]. When implementing PSPPSLL and the HA-buffer, the authors found that some necessary barriers could be left out but some redundant atomic operations had to added. Our findings agree with theirs completely. It's an unfortunate state of affairs. Fortunately, the issues can be circumvented though it's not without penalty. The redundant atomic operations slows down shader execution. Like [Lefebvre et al. 2014], we also saw significant slowdowns due to the driver issue workarounds (Section 6.3).

As we mentioned to begin with, the above-mentioned issues only occurs for configurations. In practice, we are fortunate that the issue only affected few of our test cases. However, the issues do impose unnecessary overhead due to the workaround in the cases that are affected.

3.4.4 Practicalities

Memory Allocation Until now, we have deferred the issue of allocating memory for the `data` buffer. In practice, we set the size statically, using a heuristic $|U|$ (see Section 3.3.4) based on the scene's size on disk and the available VRAM. A more scalable approach would be to allocate more memory if overflow occurred in the previous frame (similar to C++'s `std::vector`). Overflow can be detected by reading the `count` variable in the client application.

Multi-view The above-mentioned procedure describes how to construct a single layered depth map. However, we want to construct multiple layered depth maps each using a unique orientation. One solution would be to simply allocate an array of SSBOs with an entry for each layered depth map. However, OpenGL imposes implementation-defined restrictions on static arrays. Thus the solution would work but is not scalable.

Our approach is to use a single SSBO and instead provide offsets to where each layered depth map begins. A total data offset, `total_data_offset`, is used for the allocations. Initially, `total_data_offset` is zero. After constructing each layered depth map, the client application reads back `count` and adds it `total_data_offset`. I.e.,

```
total_data_offset += count + width * height
```

Note that the `total_data_offset` is also offset by `width * height` to account for the head indices. The only change to the GLSL code is the `allocate` function which now becomes

```
uniform uint32_t total_data_offset;

uint32_t allocate() {
    return total_data_offset +
        atomicCounterIncrement(count);
}
```

This change is applicable to both PPSLL and PSPPSLL construction.

An array of of offsets, `data_offsets`, records each layered depth map's individual offset into the `data` buffer. The `data_offsets` is used later to retrieve the list nodes.

Client Application Memory Barriers The client application also need a memory barrier. Specifically, a call to `glMemoryBarrier` after the layered depth maps have been created. The full call is

```
glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT)
```

which makes the SSBO contents visible to subsequent shaders.

3.4.5 Point Cloud Visualization

We now show how a layered depth map can be visualized as a point cloud. This visualization is intended to showcase how the layered depth map can be queried in practice. Simultaneously, it provides a simple middle step before we present the global illumination methods. Rendering the point cloud is done in four sub-steps:

1. **Point Splatting (screen-aligned quad).** Render each entry in the L_p as a point.
   (a) Retrieve the next depth value, `depth`, from the L_p sequence.
   (b) Reconstruct the position in WC, `wc_sample_position`, from the `depth` and the layered depth map's orientation.
   (c) Project `wc_sample_position` into the user's view.
   (d) Splat the projected point into an glsssbo.

We describe each subset in detail in the following paragraphs

(a) First, the start of the list, `current`, is found via the head node

```
// Retrieve the first node
uint32_t heads_index = data_offset + pixel_index;
uint32_t current = data[heads_index].next;
```

where `pixel_index` can be found from the fragment's xy-coordinates and `data_offset` denotes the offset into the `data` buffer (which stores the data of all layered depth maps). The list traversal itself is straightforward

```
const int max_list_length = 200;
int list_length = 0;

// Iterate the list
while (0 != current && list_length++ <
    max_list_length) {
    // Step (a)
    float depth = data[current].depth;
    // Next node
    current = data[current].next;

    // Steps (b), (c), and (d)
}
```

The `max_list_length` can be used to control the number of points generated.

(b) This step is easy since we use orthographic projection.

```
// Step (b)
vec3 view_direction = (
    forward * depth +
    right   * horizontal_scale * ndc_position.x +
    up      * vertical_scale   * ndc_position.y);
vec3 wc_sample_position = wc_view_position +
    view_direction;
```

The view vectors are given in another buffer object. The `ndc_position` variable is the current pixel's position in NDC (the pixel's position on the image plane).

(c) The projection code is straightforward but lengthy. Furthermore, it is a recurring procedure so we have put it in its own listing (Listing 4). Using said procedure, step (c) is quickly done.

```
// Step (c)
ivec2 sc_sample_position;
if (project_wc_to_sc(wc_sample_position, user_view,
    sc_sample_position)
    // Skip clipped points.
    continue;
```

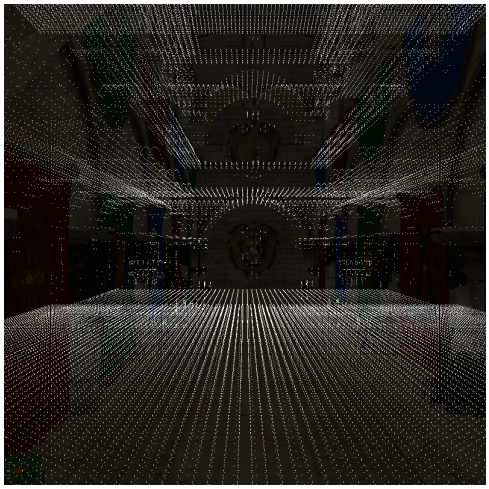
(d) Lastly, the point must be rendered. We utilize an SSBO-backed buffer, `point_cloud_image`, to store the rendered point cloud. Note that we can't simply write to the framebuffer since such writes are limited to the current pixel's coordinates. Instead we splat the `sc_sample_position` into the `point_cloud_image` buffer and let said buffer serve as our image store.

```
// Step (d)
uint32_t index = sc_sample_position.x +
    sc_sample_position.y * user_view.dimensions.x;
point_cloud_image[index] = color;
```

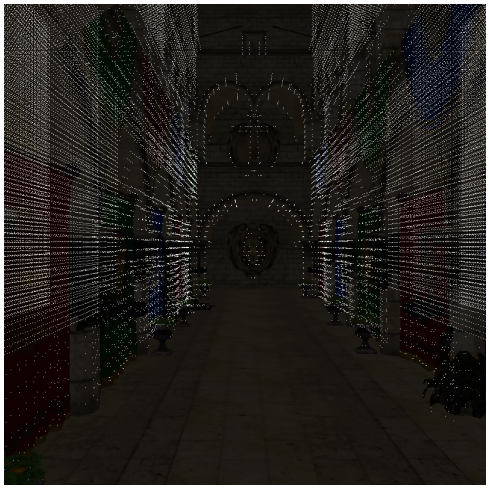
The `color` variable can be based on any metric (e.g., the depth value, the layered depth map's ID, etc.). We simply use a static color to test the visualization.

Source Code The complete GLSL source code can be found in the appendix.

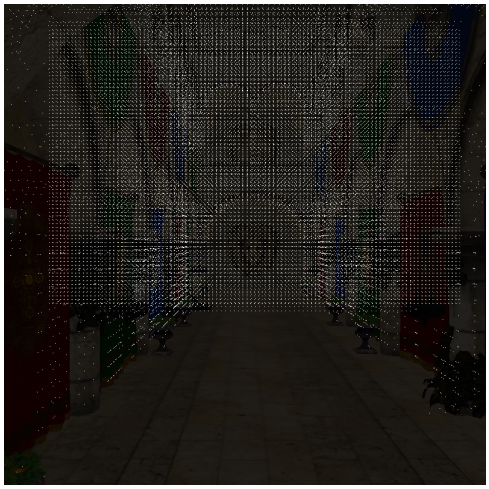
Results Figure 17 contains the point cloud visualizations of three different layered depth maps each oriented in a different direction. Note that the points are not depth-tested and thus also reveals occluded geometry. This results in an x-ray-like effect. E.g., in the top view (Figure 17a) the second floor of the sponza atrium is clearly visible. The layered depth maps use axis-aligned viewing directions. This highlights an important point: The layered depth map doesn't store fragments which are parallel to the view direction. E.g., in the side and forward view (Figures 17b and 17c), the floor is clearly not present in the layered depth maps since it is parallel with both directions. This is another reason why a multi-view approach is needed: To sample all surfaces regardless of their orientation. The three layered depth maps of Figure 17 combined provides a decent scene coverage but not individually.



(a) *Top view.*



(b) *Side view.*



(c) *Forward view.*

Figure 17: *Point cloud visualizations of layered depth maps. Each sub-figure uses a different view direction. The scene is the Crytek sponza. The point cloud rendering (white points) is overload the flat-shaded geometry.*

```
// Projects a position from world coordinates into screen coordinates corresponding to the given view. Returns
    false if the given position is outside of the user's view's bounds.
bool project_wc_to_sc( in vec3 wc_position, in view_type view, out ivec2 sc_position ) {
    // World coordinates -> Clip coordinates
    vec4 cc_position = view.view_projection_matrix * vec4(wc_position, 1.0);
    // Clipping
    if (cc_position.x > cc_position.w || cc_position.x < -cc_position.w
        || cc_position.y > cc_position.w || cc_position.y < -cc_position.w
        || cc_position.z > cc_position.w || cc_position.z < -cc_position.w)
        return false;
    // Clip coordinates -> Normalized device coordinates [-1;1]^3
    vec3 ndc_position = cc_position.xyz / cc_position.w;
    // Normalized device coordinates -> Texture coordinates [0;1]^2
    vec2 tc_position = (ndc_position.xy + vec2(1.0)) * 0.5;
    // Texture coordinates -> Screen coordinates [0;width]x[0;height]
    sc_position = ivec2(tc_position * view.dimensions);
    return true;
}
```

Listing 4: *GLSL projection routine. I.e., a transformation from WC to* screen coordinates *(SC).*

4 Ambient Occlusion

In this section, we demonstrate how our auxiliary data structure of layered depth maps can be used to compute AO. First, the background section will explain the theory behind AO. Second, previous work is presented on both real-time computation of AO. Third, we design an AO method using our auxiliary data structure. Fourth, implementation details are explained.

4.1 Background

This section will explain the mathematical model behind AO and a practical variation of said model for real-time rendering.

4.1.1 Ambient Occlusion

Recall the rendering equation [Kajiya 1986] which models light transport,

$$L_o(x,\omega_o) = L_e(x,\omega_o) + \int_{\mathcal{S}} f_s(x,\omega_o,\omega_i) L_i(x,\omega_i) \left|\cos\theta_i\right| \mathrm{d}\omega_i \quad (1)$$

where L_o is the outgoing *radiance* $\left[\mathrm{W\,m^{-2}\,sr^{-1}}\right]$ from position x in direction ω_o. The L_e term is the emitted radiance (usually from the surface of a light source). The integral is over all directions, ω_i, in the unit sphere, $\mathcal{S}$. The f term is the *bidirectional scattering distribution function* $\left[\mathrm{sr}^{-1}\right]$ which models light-surface interaction at x. The L_i term is the incoming radiance (from all the directions ω_i). Lastly, θ_i is the angle between ω_i and the surface normal, n, at x. Please refer to Figure 18 for an overview.

Equation 1 is general-purpose and can model various light transport phenomena. It is also very complex to evaluate due to the integral. Usually, Monte Carlo integration approach is used in offline rendering [Pharr and Humphreys 2004]. Such integration methods are seldom feasible in real-time rendering. Instead, assumptions are made which simplifies Equation 1 to the point that it can be easily integrated. The first such assumption is:

Assumption 1 *Surfaces do not emit light.*

Assumption 1 eliminates the L_e term from Equation 1. However, a few special surfaces must emit light since the scene rendered image would otherwise be completely dark. The solution is to treat the emitting surfaces specially and render them with another model. Furthermore, it is often the case in real-time rendering that surface emission is replaced altogether with analytical non-physical light models (e.g., point lights) [Akenine-Möller et al. 2008].

Another assumption is:

Assumption 2 *Surfaces are purely reflective.*

Assumption 2 halves the integral domain from the unit sphere, $\mathcal{S}$, to the unit hemisphere, $\mathcal{H}$, oriented in direction of the surface normal, n. That is, all directions, ω_i, which penetrate the surface at x are discarded. Only reflective directions, ω_i, remain. Again, all surfaces with scattering properties are then treated specially and rendered with another model. Furthermore, the bidirectional scattering distribution function, f_s, can be replaced with the *bidirectional reflectance distribution function* $\left[\mathrm{sr}^{-1}\right]$, f_r, since only reflection is possible.

Assumptions 1 and 2 are fairly common in real-time rendering. Another simplification is to split illumination into two groups: Direct lighting and indirect light. The former is the light which has traveled directly from the light source. The latter is all other light (e.g., light which has undergone reflection, refraction, scattering, etc.). The two groups are then modelled separately so that further assumptions can be applied individually. The following assumption are specific indirect light in the context of AO:

Assumption 3 *Surfaces are Lambertian.*

A Lambertian surface is isotropic; it scatters light equally in all directions [Pharr and Humphreys 2004]. In other words, f_r only depends on the surface position, x. Moreover, if $\rho_d(x)$ is the diffuse reflection coefficient for the surface at x, then $f_r(x) = \frac{\rho_d(x)}{\pi}$ [Pharr and Humphreys 2004]. Note that Assumption 3 implies Assumption 2. We distinguish between the two for pedagogical purposes.

The last assumption is similar to Assumption 3:

Assumption 4 *Indirect light is isotropic.*

That is, the indirect light is uniformly incident. In other words, L_i is constant in all unoccluded directions.

All assumptions combined leads to the following simplification of Equation 1

$$L_o(x,\omega_o) = \frac{\rho_d(x)}{\pi} \int_{\mathcal{H}} L_i(x,\omega_i) \cos\theta_i \mathrm{d}\omega_i \quad (2)$$

Note that the constant f_r term has been moved outside the integral and the integration domain is truncated to $\mathcal{H}$. Because of the latter, it is no longer necessary to take the absolute value of the cosine term. Now, the key to AO is to split L_i into two terms:

$$L_i(x,\omega_i) = L_i^* V(x,\omega_i) \quad (3)$$

where L_i^* is constant (due to Assumption 4). V is the so-called visibility function

$$V(x,\omega_i) = \begin{cases} 0 & \text{Ray from } x \text{in direction } \omega_i \text{intersects the scene} \\ 1 & \text{Otherwise} \end{cases} \quad (4)$$

V accounts for the occlusion so that L_i^* only contributes in all the unoccluded directions. Equation 2 can now be simplified further

$$\begin{aligned} L_o(x,\omega_o) &= \frac{\rho_d(x)}{\pi} L_i^* \int_{\mathcal{H}} V(x,\omega_i) \cos\theta_i \mathrm{d}\omega_i \\ L_o(x,\omega_o) &= \rho_d(x) L_i^* AO(x) \end{aligned}$$

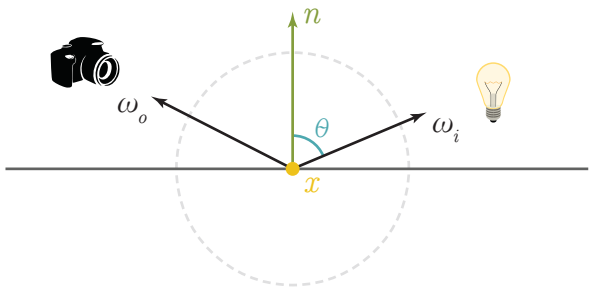


Figure 18: *The variables used in the rendering equation. Only a single direction, ω_i has been shown (pointing towards the light). Any direction in the unit sphere can be chosen for ω_i.*

where

$$AO(x) = \frac{1}{\pi} \int_{\mathcal{H}} V(x, \omega_i) \cos\theta_i \mathrm{d}\omega_i \tag{5}$$

Equation 5 is the ambient occlusion term [Zhukov et al. 1998; Landis 2002][8]. The name itself originates from older terminology. Ambient light is simply what today is mostly referred to as indirect light [Cook and Torrance 1982][9]. Today, the term ambient light is mostly used to describe indirect light under assumptions 1–4.

Note that $AO \in [0; 1]$ and is dimensionless. The unoccluded and occluded hemisphere has $AO = 1$ and $AO = 0$, respectively. This might be counter-intuitive since the term is called ambient *occlusion* and yet the definition is more naturally interpreted as ambient *visibility*. Some authors rectify this by inverting the definition to

$$AO(x) = 1 - \frac{1}{\pi} \int_{\mathcal{H}} V(x, \omega_i) \cos\theta_i \mathrm{d}\omega_i$$

or swapping the cases in the definition of V [Bavoil et al. 2008]. We do not. The form given in Equation 5 will be used for the remainder of this report.

4.1.2 Ambient Obscurance

The original formulation of Equation 5 is slightly more general [Zhukov et al. 1998]

$$AO(x) = \frac{1}{\pi} \int_{\mathcal{H}} V(x, \omega_i, d) \cos\theta_i \mathrm{d}\omega_i \tag{6}$$

where d is the distance from x to the first intersection in direction ω_i. As such, V is modified to be an attenuation function. V must abide by the following requirements:

- $V \in [0; 1]$.
- V must be monotonically increasing.

Otherwise, V can be chosen freely. A third requirement can be added to limit the maximum extent of V:

- $V(\cdot, \cdot, d_{max})$ must return 1 beyond a certain distance, d_{max}.

The last requirement is popular in real-time rendering (Section 4.2.3). This is because the tracing radius can be limited to a finite distance. Note that $V(x, \omega_i)$ is a special case of $V(x, \omega, d)$ where $d_{max} = \infty$. When V is attenuated, the term in Equation 6 is known as *ambient obscurance* (AO). Note the ambiguous acronym.

The attenuated version of V is not intended to be physically correct. It is merely used to overcome some practical limitations of the unattenuated version. Specifically, to limit the extent of V so that intersection tests can be limited to a finite search radius. Moreover, the attenuated version gives additional control to artists and can be tweaked according to aesthetics. Again, we want to stress that this added control is not physically based.

Lastly, it should be noted that some authors use the terms ambient occlusion and ambient obscurance interchangeably. We follow that convention and let the context solve the ambiguity. I.e., whether $V(x, \omega_i)$ or $V(x, \omega_i, d)$ is used.

[8] The general model (Section 4.1.2) was invented by [Zhukov et al. 1998]. The term ambient occlusion was coined by [Landis 2002].

[9] In fact, [Cook and Torrance 1982] presents a term called f which is the precursor to the modern AO definition.

4.2 Previous Work

This section will explain how the integral in Equation 6 has been computed previously. First, we describe a simple solution which used to be popular in real-time rendering. Second, we describe the Monte Carlo estimator. Third, we look into screen-space methods. Fourth, methods related to layered depth maps are presented.

4.2.1 Constant Ambiance

The simplest solution is simply to use a constant AO term. This approach avoids computing the integral in Equation 5 altogether which is the cheapest option performance-wise. As such, it has been used in real-time rendering for a long time [Akenine-Möller et al. 2008]. However, the result is flat-shaded surfaces since the model lacks any kind of directionality.

4.2.2 Monte Carlo Integration

An alternative is to choose sample directions, ω_i, which represents the hemisphere $\mathcal{H}$. As mentioned earlier, this is often done using Monte Carlo integration in offline use. The Monte Carlo estimator [Pharr and Humphreys 2004] is

$$\int g(x)\mathrm{d}x \approx \frac{1}{N}\sum_{i=1}^{N}\frac{g(X_i)}{p(X_i)} \tag{7}$$

where N is the number of samples and X_i is a sample from the integration domain. The *probability density function* (PDF), $p(X)$, denotes the probability for X to be chosen from the integration domain. The simplest PDF is for the uniform distribution where all samples are equally probable to be chosen. For choosing directions on the unit hemisphere, $\mathcal{H}$, said PDF is

$$p(\omega) = \frac{1}{A_{\mathcal{H}}} = \frac{1}{2\pi}$$

where $A_{\mathcal{H}} = 2\pi$ is the area of the unit hemisphere. Applying Equation 7 to the AO term of Equation 6 yields that

$$\begin{aligned} AO(x) &\approx \frac{1}{\pi N}\sum_{i=1}^{N}\frac{V(x,\omega_i,d)\cos\theta_i}{p(\omega_i)} \\ &= \frac{A_{\mathcal{H}}}{\pi N}\sum_{i=1}^{N}V(x,\omega_i,d)\cos\theta_i \\ &= \frac{2}{N}\sum_{i=1}^{N}V(x,\omega_i)\cos\theta_i \end{aligned} \tag{8}$$

where each direction, ω_i, is sampled from the unit hemisphere uniformly at random[10]. The intersection test in V can be implemented with ray-tracing [Landis 2002].

Importance Sampling Alternatively, one can sample from the cosine-weighted hemisphere [Pharr and Humphreys 2004]. The PDF for the latter is

$$p(\omega) = \frac{\cos\theta}{\pi}$$

where θ is the angle between ω and the normal which defines the hemisphere. This sampling strategy is known as importance sampling. The idea is to choose a PDF which has similar properties to the function g in Equation 7. The purpose is to reduce variance in the result. In the case of AO, importance sampling can also simplify the computation. E.g., for the cosine-weighted hemisphere

$$\begin{aligned} AO(x) &\approx \frac{1}{\pi N}\sum_{i=1}^{N}\frac{V(x,\omega_i,d)\cos\theta_i}{p(\omega_i)} \\ &= \frac{1}{N}\sum_{i=1}^{N}V(x,\omega_i,d) \end{aligned} \tag{9}$$

Note that both π and the cosine term cancel out.

4.2.3 Screen-space Methods

The aforementioned method uses ray-tracing to evaluate V. Thus it is not directly applicable to rasterization. Instead, so-called screen-space (or image-space) methods are used. Said methods typically use the depth map as a coarse scene representation from which global information can be queried [Shanmugam and Arikan 2007; Mittring 2007]. These methods are known as SSAO methods.

Point Sampling The initial approach was to estimate the AO term in Equation 6 using the following approximation [Shanmugam and Arikan 2007; Mittring 2007]

$$AO(x) \approx \frac{1}{N}\sum_{i=1}^{N}V^*(x_i,d_i) \tag{10}$$

where x_i are sample positions chosen uniformly at random in a sphere of radius d_{max} around x. d_i is the distance between x_i and x. There are many noteworthy differences to the analytical formulation of AO. First, all directionality has been taken out of the problem. Consequently, this approximation is missing the cosine term which usually weighs each sample. Second, the visibility function, V, has been replaced with V^*

$$V^*(x) = \begin{cases} 0 & \text{If } x \text{is occluded according to the depth map} \\ 1 & \text{Otherwise} \end{cases}$$

In practice, the occlusion test is performed by comparing x's depth value with the corresponding depth value in the depth map. Note that V^* is not attenuated. Instead, d_{max} is used to control the spread of the samples. A small spread localizes depth map accesses and thus increases performance. Third, samples are chosen in sphere and not a hemisphere as dictated by Equation 6. Thus half of the samples are expected to be occluded even on a flat surface. To compensate, $\frac{1}{2}$ is added to the result.

Due to the above points, Equation 10 is a very coarse approximation of the AO of Equation 5. As such, it is best interpreted as an artistic interpretation of AO. Still, SSAO is used in practice because of its good performance characteristics [Mittring 2007].

Horizon Sampling Another screen-space approach approximates AO as a function of the unoccluded horizon [Bavoil et al. 2008]. It is a high quality screen-space approximation so we will derive it here. First of all, the definition of Equation 6 must be double inverted

$$AO(x) = 1 - \frac{1}{\pi}\int_{\mathcal{H}}(1 - V(x,\omega_i))\cos\theta_i\mathrm{d}\omega_i$$

Note that equality still holds (see the appendix for a short proof). The first approximation is

$$AO(x) \approx 1 - \frac{1}{2\pi}\int_{\mathcal{H}}(1 - V(x,\omega_i))\,\mathrm{d}\omega_i$$

[10] Note that the subscript i is now used to index the sample. Beforehand, the i was used to indicate that ω was the *incoming* direction.

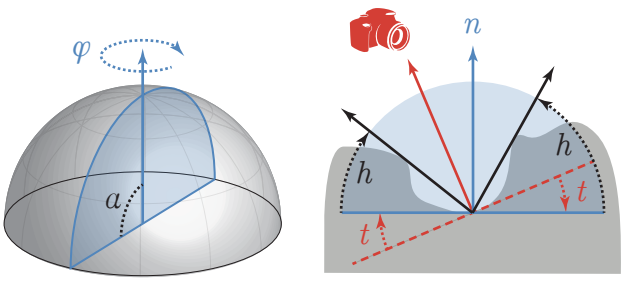


Figure 19: *Overview of the HBAO method.*

Notably missing is the cosine term inside the integral which is compensated for by the division with two. Again, this coarse approximation is best interpreted as an artistic interpretation. The next step is to split the integral over directions into two integrals over the corresponding spherical coordinates

$$AO(x) \approx 1 - \frac{1}{2\pi} \int_{\phi=0}^{2\pi} \int_{\alpha=0}^{\frac{\pi}{2}} (1 - V) \cos\alpha \mathrm{d}\alpha \mathrm{d}\phi$$

To each ϕ (the azimuth angle) corresponds a 2D slice of the hemisphere. The inner integral using α (the elevation angle, $\alpha = \frac{\pi}{2} - \theta_i$) is within said slice. Thus the inner integral can be interpreted as a measurement of the occluded horizon within said slice. Occluded, since the inner integral is over $(1 - V)$ and $V = 0$ denotes occluded directions. Now, assume that the occluded horizon is bounded above $t < \alpha < h$ for some t and h. Thus it must be that the $(1 - V)$ is 0 outside those bounds. This information simplifies the inner integral to

$$\begin{aligned} AO(x) &\approx 1 - \frac{1}{2\pi} \int_{\phi=0}^{2\pi} \int_{\alpha=t(\phi)}^{h(\phi)} \cos\alpha \mathrm{d}\alpha \mathrm{d}\phi \\ &= 1 - \frac{1}{2\pi} \int_{\phi=0}^{2\pi} (\sin(h(\phi)) - \sin(t(\phi))) \, \mathrm{d}\phi \end{aligned}$$

Note that t and h are functions of ϕ since they depend on how the hemisphere is sliced. The last approximation is to use the Monte Carlo estimator (Equation 7) for the remaining integral. Azimuth directions are chosen uniformly at random so $p(\phi) = \frac{1}{2\pi}$. The integral then becomes

$$\begin{aligned} AO(x) &\approx 1 - \frac{1}{2\pi N} \sum_{i=1}^{N} \frac{\sin(h(\phi_i)) - \sin(t(\phi_i))}{\frac{1}{2\pi}} \\ &= 1 - \frac{1}{N} \sum_{i=1}^{N} \sin(h(\phi_i)) - \sin(t(\phi_i)) \qquad (11) \end{aligned}$$

where N is the number of samples. Equation 11 models what is known as *horizon-based ambient occlusion* (HBAO).

In the aforementioned derivation, t and h were just assumed to be known. Theoretically, the horizon should be bounded between $0 < \alpha < \alpha_{horizon}$ for some horizon angle $\alpha_{horizon}$. In practice, however, sampling for the upper bound is done in screen-space (i.e., in the view plane). Thus the bounds must be offset to $t < \alpha < h$ to compensate. t is be the angular offset from the view plane (defined by the view direction) to the surface plane (defined by the normal at x). h is found via ray-marching the depth map in direction ϕ. Let the number of ray-marching steps be S and the marching distance be d_{max}. Thus the method depends on three parameters: N, S, and d_{max}. Note how d_{max} has been worked into the approach in order to limit the extent of the ray-marching. Please refer to Figure 19 for an overview.

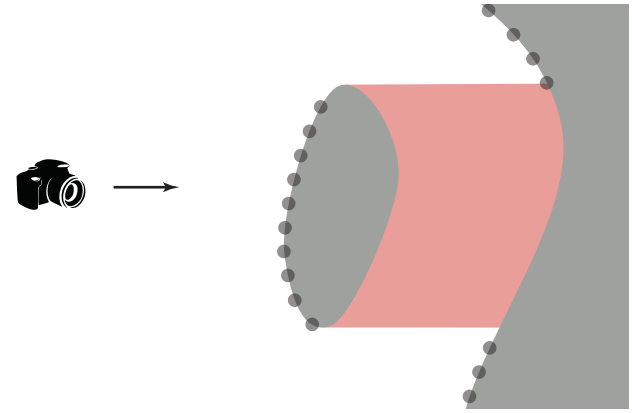


Figure 20: *Breakdown of the continuous height field assumption. The depth values (circles) stored in the depth map do not adequately represent the scene. The area shown in red is assumed to be occluded even though it is not.*

Analogously, the attenuated version of V can be used. There is also an earlier horizon-based approach where the cosine term is not approximated away [Dimitrov et al. 2008]. This approach, however, has seen little practical application.

Summary There are many other variations of SSAO [Filion and McNaughton 2008; Loos and Sloan 2010; McGuire et al. 2011; Mittring 2012]. Common for them all is that they fit within a real-time render budget both in terms of performance and memory use. We have similar requirements for our AO model. Therefore, we will implement one screen-space approach so that we can compare it to our own method. Specifically, we will implement HBAO since it close (relatively few approximations) to the original definition of AO.

4.2.4 Using Layered Depth Maps

The screen-space methods mentioned before all used the depth map as an auxiliary data structure. This has notable limitations. Specifically, the height-field is assumed to be continuous [Bavoil et al. 2008]. This assumption fails when an increase in the depth values between two pixels is not due to steep geometry but due to spatial disconnection (Figure 20). In other words, the depth map is a very coarse scene representation. Instead, a layered depth map can be used. It does not require the continuous height-field assumption since it also captures occluded geometry.

Extending the Screen-space Methods One approach is to use the model of an SSAO approach as a basis but using a layered depth map for lookups [Shanmugam and Arikan 2007; Bavoil and Sainz 2009; Bauer et al. 2013; Liu et al. 2013]. Any basis model is applicable. We'll use Equation 10 as an example. First, a single layered depth map is constructed from the user's view direction instead of adepth map. The key difference is that V^* now iterates through all layers of the layered depth map in order to determine whether a sample is occluded. Contrast this to the regular approach where only the first layer is tested.

In conclusion, the layered depth map provides more geometric information compared to the depth map. The downside is the added

performance cost due to both the construction and sampling of the layered depth map [Bauer et al. 2013]. Another advantage is that the AO can be combined with OIT in a hybrid method [Bauer et al. 2013].

Ray-tracing Methods During the discussion of SSAO methods, we hinted that computing V with ray-tracing was limited to offline methods. Fortunately, this is not the whole truth. From the discussion in Section 3.3, we already now that layered depth map can be used for intersection queries. Moreover, there already exist hybrid approaches which combine rasterization with elements of ray-tracing using layered depth maps. These hybrids are not limited to AO but can solve many different light transport problems. Therefore, we defer the discussion of these methods to Section 5.2. We will, however, hint at how this can be done in the next section. Thus said section serves as a middle step before we explore more complex approaches.

4.3 Design

In this section, we will design a method to compute AO in rasterization with layered depth maps. First, we connect the theory from Section 3.3 with the theory of AO. Second, we explain our sampling strategy. Third, AO is combined with environment lighting.

4.3.1 Layered Depth Maps and AO

Recall that a layered depth map can be used to perform intersection queries in the direction which the layered depth map is oriented (Section 3.3). This is exactly the kind of query that is needed to implement the visibility function V. Formally, Equation 4 can be implemented as

$$V(x, \omega_i) = \begin{cases} 0 & \text{trace}(x, \omega_i)\text{returned a position} \\ 1 & \text{Otherwise} \end{cases} \tag{12}$$

where we have used the previously mentioned trace function. The problem is that $V(x, \omega_i)$ is defined for multiple directions and not just a single one. Our solution is to construct multiple layered depth maps each oriented in a different direction. Section 4.3.2 will go into further details.

4.3.2 Sampling Strategy

The sampling strategy dictates how the directions, ω_i, are chosen. Select strategies are explained in the following paragraphs.

Random One approach is to use the exact same strategy as done in ray-tracing. I.e., solve the integral using the Monte Carlo estimator (Equation 8) and therefore choose the sample directions uniformly at random. Said directions could be generated each frame since our auxiliary data structure reconstructs all the layered depth maps on a per-frame basis. The advantage of random sampling is that the result is free of banding artifacts (though some noise is introduced) [Pharr and Humphreys 2004]. There are, however, some caveats to this approach. The set of directions is very likely to be different from frame to frame (since the directions are generated randomly). Consequently, the sampling pattern will change each frame. This results in temporal flickering (flickering noise between frames).

The obvious solution is to instead generate the random directions at application start (and not before each frame). This approach may indeed work well for a large number of sample directions. In practice, however, we would like to limit the number of sample directions in order to increase performance. This is especially true for our auxiliary data structure since each sample direction requires the generation of a layered depth map. Moreover, the directions generated at application start may not cover the hemisphere sufficiently (if we are unlucky with the random generation). In that case, all subsequent frames will be stuck with a poor sample distribution. The fewer samples that are used, the likelier it is that the sample distribution will be poor.

Lastly, importance sampling is not applicable due to the nature of our auxiliary data structure. When using importance sampling in the context of AO (e.g., Equation 9), the PDF is based on surface properties (e.g., the normal). As such, it is impossible to construct the sample directions prior to rasterizing the scene since the surface properties are unknown at that time.

Quasi-random The aforementioned caveats are well-known problems in offline rendering. A solution is to use quasi-random samples instead. That is, samples which are chosen by a combination of random sampling and deterministic sample distribution. One such strategy is stratified sampling [Pharr and Humphreys 2004]. Let N be the total amount of samples. First, the sample domain (e.g., the unit hemisphere), Λ, is divided into subdomains, $\Lambda_1, \Lambda_2, \ldots, \Lambda_n$. Second, an equal amount of samples, $\frac{N}{n}$, are generated at random in each subdomain, Λ_i. Since the randomness is restricted to the subdomains, samples are less likely to group up in clusters. Thus even with a low number of overall samples, N, an even distribution of samples can be guaranteed.

Using quasi-random sampling with the Monte Carlo estimator is known as the quasi-Monte Carlo method [Pharr and Humphreys 2004]. The number of subdomains, n, determines the trade-off between banding artifacts (due to determinism) and noise (due to randomness).

The above properties make quasi-random sampling a good candidate for our purpose. This is regardless of whether the samples are generated per frame or at application start. Still, the caveats mentioned about pure random sampling remain though they are now controllable via the parameter n.

Deterministic Lastly, is the option to choose sample directions deterministically. As hinted above, deterministic strategies produce no noise but suffers from banding artifacts.

In the case of the unit hemisphere, one way to divide the domain is by equal area. The sample directions are then the centers of each subdomain. Equal-area subdivision, however, can be achieved using ring slices (and a spherical cap). With such a subdivision, it is impossible to choose a proper center for each subdomain (apart from the cap). Therefore, it is additionally required that each subdomain must have a small diameter. The latter is defined as the maximum Euclidean distance between two points in the domain. Combined, the equal-area and small-diameter requirements restrict the subdivison to well-distributed patches from which representative centers can be easily chosen.

Such a subdivision has been achieved for the sphere through what is known as the recursive zonal equal-area partition [Leopardi 2006]. Said algorithm recursively divides the domain into equal-area small-diameter subdomains starting with the entire sphere. It returns both the subdomains and their centers (which we interpret as directions). Please refer to Figure 21 for an example. The complete algorithm is complex and out of scope of this report.

The subdivision of [Leopardi 2006] is for the sphere. It can, however, be easily modified to work for the hemisphere defined by the normal n. The solution is simply to reject directions which are in the complementary hemisphere. E.g., directions, ω, for which $\cos(\omega \cdot n) < 0$.

An additional modification is required for our use case. Recall that a layered depth map oriented in direction d can test for intersections in both the d and $-d$ directions (Section 3.3). As such, only a single layered depth map is needed for all parallel directions. Therefore, we further cull the sample directions so that no two directions are parallel.

The equal-area small-diameter distribution of directions ensures that the hemisphere is representatively sampled. Analogously, each direction, ω, can be interpreted as representing its corresponding subdomain of the hemisphere, A_{sub}. Formally, ω is interpreted as a differential cone of directions with solid angle $\mathrm{d}\omega = \frac{A_{\text{sub}}}{r^2}$ where r is the radius of the hemisphere. Thus $\mathrm{d}\omega = A_{\text{sub}}$ on the unit hemisphere. As such, the integral in Equation 5 can be approxi-

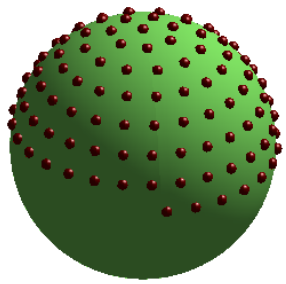


Figure 21: *Visualization of 128 samples generated using the recursive zonal equal-area partition [Leopardi 2006]. Note that half of the samples in the bottom ring have intentionally been culled because parallel sample directions exist on the other side of the sphere.*

mated using the midpoint rule

$$AO(x) \approx \frac{1}{\pi}\sum_{i=0}^{N} V(x,\omega_i)\cos\theta_i A_{\text{sub},i}$$

where $A_{sub,i}$ is the area of the ith subdomain and N is the total number of subdomains. Since all subdomains have equal area, it must be that

$$A_{\text{sub},i} = \frac{A_{\mathcal{H}}}{N} = \frac{2\pi}{N}$$

where $A_{\mathcal{H}} = 2\pi$ is the area of the unit hemisphere. Thus the integral becomes

$$AO(x) \approx \frac{2}{N}\sum_{i=0}^{N} V(x,\omega_i)\cos\theta_i \tag{13}$$

Note that this coincides with Equation 8 where uniform random sampling was used. Again, we stress that the deterministic equal-area small-diameter approach introduces banding artifacts.

Alternatively, the deterministic sampling method could have been prioritized according to the cosine term. In our use case, however, this is not applicable for the same reasons as earlier noted with importance sampling.

Summary We use the deterministic approach to ensure consistency between frames (temporal coherence). In doing so, we acknowledge that the results will have banding artifacts. The reasoning is that the flickering noise of random or even quasi-random sampling strategies is too much of a disturbance. We fear that it might ruin the user experience. Banding artifacts are also noticeable but a much minor distraction.

4.3.3 Environment Lighting

So far, AO has been explained as a special case of indirect lighting. However, the underlying assumptions made can also be applied to direct environment lighting. Specifically, Assumption 4 is true for distant, omnidirectional light such as that of a cloudy sky environment. In this context, L_i^* (from Equation 3) denotes the average color of the environment. However, if the environment is non-uniform (e.g., due to one big bright spot such as the sun), then Assumption 4 doesn't hold. Alternatively, one can choose a different simplification of L_i such as

$$L_i = L_{env}(\omega_i)V(x,\omega_i)$$

where $L_{env}(\omega_i)$ is the incoming radiance from the environment in direction ω_i. Note that $L_{env}(\omega_i)$ is assumed to not depend on x. I.e., the environment is assumed to be so far away that any change with regard to x will be infinitesimal in comparison. The result is environment occlusion

$$EO = \frac{1}{\pi}\int_{\mathcal{H}} L_{env}(\omega_i)V(x,\omega_i)\cos\theta_i \mathrm{d}\omega_i$$

which fits into the rendering equation similarly to AO

$$L_o(x,\omega_o) = \rho_d(x)EO(x)$$

In practice, $L_{env}(\omega_i)$ is implemented via environment mapping. Furthermore, lookups into the environment map can be dependent on the result of V to increase performance.

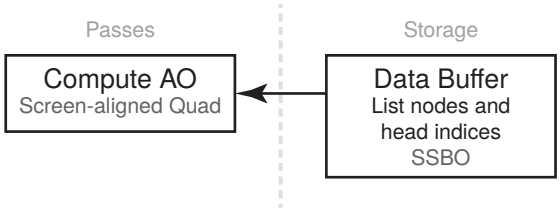


Figure 22: *The passes and storage used to generate AO.*

4.4 Implementation

In this section, we will show how our AO method can be implemented in practice. First, the sampling method is implemented. Second, a practical issue is solved using a normal offset.

4.4.1 AO Sampling

The sampling directions used in Equation 13 are pre-computed using a Matlab script provided by [Leopardi 2006]. Said script has been modified according to the discussion in Section 4.3.2. When the application starts, the directions are loaded.

Solving Equation 13 requires the following steps (Figure 22):

1. **Compute AO (screen-aligned quad).** Choose a sample direction w_i.
   (a) Compute trace(x, ω_i) where x is the position in WC corresponding to the current pixel.
   (b) V is calculated using Equation 12.
   (c) Weigh V by the cosine term using the normal from the G-buffer and accumulate the result.

Note that the accumulated result must be weighed by $\frac{2}{N}$ to get the final result. We will go into further detail with each sub-step in the following paragraphs.

(a) We already gave an overview of the trace(x, ω_i) computation in Section 3.3. Now, we will provide an implementation. First (**Find Map**), the layered depth map is corresponding to ω_i found. This is trivial, since there is a direct mapping between the two. Second (**Find Pixel**), x is projected into the view of the layered depth map to find the corresponding pixel, p

```
// Find the pixel's position
ivec2 sc_position;
if (project_wc_to_sc(wc_position, ldm_view,
    sc_position)
    // Assume clear outside of LDM bounds
    return 1.0;
```

Note that the `project_wc_to_sc` function from Listing 4 has been reused. Also, if the position is clipped by the projection, then x must have been outside the view used to generate the layered depth map. Such positions are assumed to belong to the environment.

Third (**Find Depth Value**), the list is traversed. `sc_position` can be used to index into the `data` buffer of the layered depth map

```
// Get the head node
uint32_t head_index = data_offset + sc_position.x +
    sc_position.y * ldm_view.dimensions.x;
uint32_t current = data[head_index].next;
```

to find the corresponding head node of the singly linked list (L_p sequence). As explained earlier, the `data_offset` is due to all the layered depth maps being stored in the same buffer. Traversing said list is is identical to the code used for the point cloud visualization (Section 3.4.5)

```
// Traverse the list
while (0 != current && list_length++ <
    max_list_length) {
    wc_sample_position = /* Same as before */

    // Find list node corresponding to x

    // Next node
    current = data[current].next;
}
```

The same goes for the reconstruction of the sample position, `wc_sample_position`, from the stored depth value . The new part is to find the list node which corresponds to x (`wc_position`). Said list node must be the one for which `wc_position == wc_sample_position`. In practice, however, such a comparison will always fail due to finite precision in computing. Instead, we find the list node for which `sample_distance`

```
float sample_distance = distance(wc_sample_position,
    wc_position);
```

is minimum. This is done using a simple if statement

```
// Find list node corresponding to x
if (sample_distance < min_distance) {
    // Store data about the previous node.
    // Just the distance is needed for AO.
    previous_distance = min_distance;

    min_distance = sample_distance;
// Diverging
} else break;
```

where `min_distance` is initially the largest floating point value (`FLOAT_MAX`). Because the singly linked list is sorted, the loop can be broken as soon as the `sample_distance` starts to increase (diverges). After the loop, the `current` node will correspond to x.

Fourth (**Compute Intersection**), the previous node (corresponding to z_{x-1}) has already been found. In the context of AO, only the distance to the previous node, `previous_distance`, is needed. For other methods, any additional data about the previous node can also be saved if need be.

(b) Most of the work is already done at this point. V is implemented as follows

```
float visibility( in float occluder_distance )
{ return (FLOAT_MAX == occluder_distance) ? 1.0 :
    0.0; }
```

and called like this

```
float V = visibility(previous_distance);
```

Alternatively, an attenuated version of V can be used

```
const float d_max = 100.0;
const float falloff_exponent = 2.0;
float attenuated_visibility( in float
    occluder_distance )
```

```
float trace_ambient_occlusion(
    in vec3 wc_position,
    in vec3 wc_normal )
{
    float result = 0.0;
    for (int i = 0; i < N; ++i)
        result += trace_ambient_occlusion(
            i,
            wc_position,
            wc_normal);
    return 2.0 / float(N) * result;
}
```

Listing 6: *GLSL summation of AO contributions.*

```
{ return pow(min(occluder_distance / d_max, 1.0),
    falloff_exponent); }
```

to implement ambient obscurance.

(c) Likewise, the cosine term is easily evaluated

```
float cos_theta = dot(ldm_view.forward, wc_normal);
```

Lastly, the contribution is accumulated

```
float result += V * cos_theta
```

Source Code The complete GLSL code for a fragment shader implementing the above steps can be found in Listing 5. The last step is to sum all the contributions together as shown in Listing 6. Note that in the complete code, the distance to the next link node, `next_distance`, is also computed. The AO contribution is only computed for one sample. Namely the one which is in the unit hemisphere defined by `wc_normal`.

4.4.2 Normal Offset

In practice, the above implementation produces artifacts for thin surfaces. The problem is that the algorithm sometimes mistakes the first occluder (corresponding to z_{x-1}) for x itself at oblique angles. This is an artifact of the low resolution of the layered depth map.

The issue can be resolved by introducing a normal offset. That is, by virtually offsetting the positions along the normal and thereby thickening the surfaces. The normal offset is applied as follows

```
// Normal offset (virtual thickening)
float cos_alpha = clamp(dot(wc_normal, ldm_view.
    forward), 0.0, 1.0);
float normal_offset = sqrt(1.0 - cos_alpha *
    cos_alpha); // sin(acos(cos_alpha));
const float constant_factor = 10.0;
wc_position += wc_normal * normal_offset *
    constant_factor;
```

Note that the offset is weighed by $\sin(\mathrm{acos}(\omega_i \cdot n))$ where ω_i is the direction of the layered depth map and n is the surface normal. The more the n deviates from ω_i the greater the offset. This is inspired by a similar technique used to reduce artifacts in shadow mapping [Holbert 2011].

4.4.3 HBAO

The source code to our HBAO implementation can be found in the appendix. The implementation is based on our previous work [Aalund and Bærentzen 2013]. Usually, a post-processing blur is used to remove noise artifacts in screen-space methods [Loos and Sloan 2010; McGuire et al. 2011]. The HBAO does not need to be blurred if enough samples are taken. As such, HBAO can be implemented in a single pass over a screen-aligned quad. Sampling is done from an attached depth map which is rendered during G-buffering.

```
float trace_ambient_occlusion( in int ldm_id, in vec3 wc_position, in vec3 wc_normal ) {
    // Get LDM data
    view_type ldm_view = views[ldm_id];
    uint32_t data_offset = data_offsets[ldm_id];

    /* Normal offset */

    // Find the pixel's position
    ivec2 sc_position;
    vec3 ndc_position;
    if (project_wc_to_sc(wc_position, ldm_view, sc_position, ndc_position)
        // Assume clear outside of LDM bounds
        return 1.0;

    // Get the head node
    uint32_t head_index = data_offset + sc_position.x + sc_position.y * ldm_view.dimensions.x;
    uint32_t current = data[head_index].next;

    // Initialize search variables
    float min_distance = FLOAT_MAX;
    float previous_distance = min_distance;
    float next_distance = min_distance;
    bool get_next = false;

    // Traverse the singly linked list
    const int max_list_length = 2048;
    int list_length = 0;
    while (0 != current && list_length++ < max_list_length) {
        float depth = data[current].depth;

        // Reconstruct the position in world coordinates
        vec3 direction = (
            ldm_view.forward * depth +
            ldm_view.right   * ldm_view.horizontal_scale * ndc_position.x +
            ldm_view.up      * ldm_view.vertical_scale   * ndc_position.y);
        vec3 wc_sample_position = ldm_view.eye + direction;

        float sample_distance = distance(wc_sample_position, wc_position);

        // Keep track of the next node in the list
        if (get_next) {
            get_next = false;
            next_distance = sample_distance;
        }

        // Keep track of the previous node in the list
        if (sample_distance < min_distance) {
            previous_distance = min_distance;
            min_distance = sample_distance;
            get_next = true;
        } else break;

        current = data[current].next;
    }
    if (get_next)
        next_distance = max_distance;

    float cos_theta = dot(ldm_view.forward, wc_normal);

    return (cos_theta > 0.0)
        ? visibility(next_distance) * cos_theta
        : visibility(previous_distance) * -cos_theta;
}
```

Listing 5: *GLSL computation of AO using layered depth maps.*

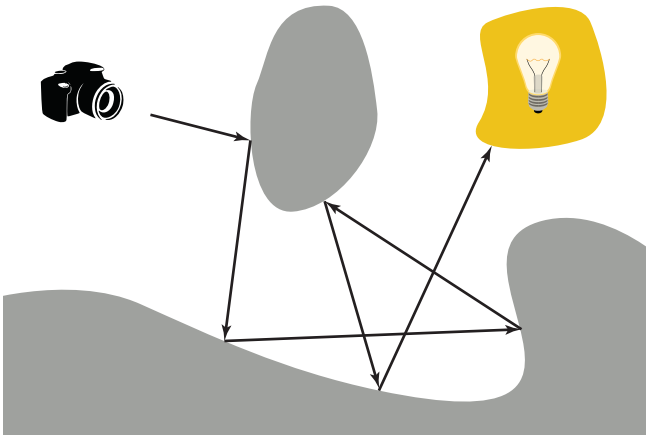


Figure 23: *Path tracing. A path is traced from the camera to the light source. At each surface intersection, a new random direction is randomly sampled.*

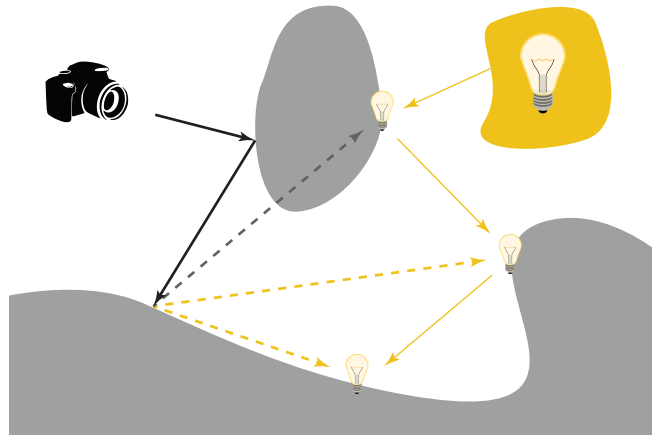


Figure 24: *Virtual point light. First, a light path is traced (yellow arrow). At each vertex, a VPL is generated (small light bulbs) which represents the light path thus far. Second, a camera path is traced (black arrow) which samples all the point lights for each vertex (dashed arrows). Occluded lights do not contribute (black dashed arrows).*

5 Indirect Lighting

In this section, we present a global illumination method for single-bounce indirect diffuse lighting using our auxiliary data structure of layered depth maps. First, a theoretical introduction to indirect lighting is given. Second, the previous work section presents an overview real-time indirect lighting methods. Third, we present our indirect lighting method based on photon differentials and using layered depth maps. Fourth, implementation details are given.

5.1 Background

This section describes various methods to produce indirect lighting. All methods are derived from the rendering equation. First, we describe path tracing and VPLs. These methods will be used in our comparison. Then, we go into details with photon mapping and photon differentials. The latter will be the basis of our approach.

5.1.1 Path Tracing

Path tracing was presented together with the rendering equation (Equation 1) as a general-purpose method to solve the latter [Kajiya 1986]. In Section 4.1, the rendering equation was used to derive a simplified version of indirect lighting known as AO. This was done under Assumptions 1–4. Such assumptions are a useful to simplify the integral but not strictly necessary. Using path tracing, the integral can be computed in its unsimplified form. That is, path tracing is general global illumination method for both direct and indirect light. The idea is to trace a path from the eye (E) to a light source (L) in the scene. Said path can undergo any number of diffuse (D) or specular (S) surface interactions. That is, any $L(D|S)^*E$ path in light transport notation [Heckbert 1990].

First, a ray is traced from the eye to compute the first intersection point. A new ray is then traced from the intersection point in a direction sampled on the unit sphere, $\mathcal{S}$. In practice, the Monte Carlo method is used to sample directions with a PDF based on the surface's BSDF (importance sampling). New rays are traced recursively until either: The ray hits a light, Russian roulette terminates the ray, or a max tracing depth has been reached [Pharr and Humphreys 2004]. Russian roulette is used to probabilistically stop the tracing without introducing bias. All traced rays combined form the path (Figure 23).

With all the ray-tracing in the algorithm, it can be hard to distinguish path tracing from conventional Whitted ray-tracing [Whitted 1980]. The key difference is that Whitted ray-tracing forms a tree of rays whereas path tracing forms a single path of rays [Kajiya 1986].

Bidirectional Path Tracing Obscured light sources pose a problem in conventional path tracing. If the light is hard to reach from the camera, then many paths will fail to find it and thus not contribute to the final image. By also constructing paths from the light sources, otherwise obscured light paths can easily be found. This is known as bidirectional path tracing [Pharr and Humphreys 2004]. Paths from the camera and light sources are connected by visibility rays. This method is better at handling obscured light sources. It has similar performance characteristics to path tracing.

Summary The key advantage to path tracing is that it is an unbiased method. As such, it is known to converge to the correct solution if given enough time. One of the disadvantages is that it may take a substantial amount of time before the solution converges. Too few paths results in under-sampling and, consequently, noise. Moreover, the algorithm uses ray-tracing to construct the paths and as such doesn't directly apply to a rasterization-based pipeline. With layered depth maps, however, this limitation can be overcome. See Section 5.2 for a path tracing-rasterization hybrid based on layered depth maps.

5.1.2 Virtual Point Lights

Correctness can be traded for performance. This is the key to methods which use approximations and estimates to solve the rendering equation. Such methods are biased since they will never converge to the true solution in practice. However, a biased solution can be visually convincing nonetheless.

One such method is instant radiosity [Keller 1997][11]. First, the ren-

[11] Instant radiosity is based on the radiosity method [Goral et al. 1984].

dering equation must be converted into an integral over surface area. The relation between differential solid angle and differential area is

$$\mathrm{d}\omega = \frac{\cos\theta'}{r^2}\mathrm{d}A$$

where r is the distance between x and $\mathrm{d}A$. θ' is the angle between the surface normal at $\mathrm{d}A$ and ω [Pharr and Humphreys 2004]. Using the above relation, the rendering equation can be written as

$$L_o(x,\omega_o) = L_e(x,\omega_i) + \int_A f_r(x,\omega_o,\omega_i)L_i(x,\omega_i)V(x,\omega_i)\cos\theta\frac{\cos\theta'}{r^2}\mathrm{d}A(y)$$

where A is the surface area of the entire scene, y is a point representing $\mathrm{d}A$, and ω_i is the direction from x to y. The visibility term is the same as used in AO. A small simplification is often used

$$L_o(x,\omega_o) = L_e(x,\omega_i) + \int_A f_r(x,\omega_o,\omega_i)L_i(x,\omega_i)G(x,y)\mathrm{d}A(y) \quad (14)$$

where the geometry term

$$G(x,y) = V(x,\omega_i)\frac{\cos\theta\cos\theta'}{r^2}$$

accounts for distance attenuation and projection between the surfaces. Now, the basic idea in instant radiosity is to sample the incoming radiance, L_i, from so-called *virtual point light* (VPLs). The method uses two passes:

1. **VPL Generation.** Light paths are constructed from the light sources. Each time a vertex is added to the light path, a VPL is stored. Said VPL represents the light path up until that point.
2. **Rendering.** Rendering is done in two parts. Direct lighting is sampled using another method (e.g., Whitted ray-tracing). Indirect lighting is computed by connecting the camera paths with the light paths through the VPLs.

Note that the light path is limited to diffuse reflections, LD^+, because of Assumption 3. Each pass will be explained in more detail in the following paragraphs. Please refer to Figure 24 for an overview.

Generation Each VPL stores a representation, α, of the corresponding light path. The initial VPL's α is based on the radiance emitted from the light (L_e). Each subsequent VPL's α is based on the incoming radiance from the previous vertex in the light path. The radiance is scaled between the vertices to account for projection (cosine term) and and the BRDF at the current vertex. Specifically, α is the product of L_e with the the cosine term (for each vertex) and the BRDF (for each vertex beyond the first). Additionally, each VPL stores the properties of there surface where the VPL was generated (the position, normal, and bidirectional reflectance distribution function).

In theory, α is just an intermediary variable used to store the light path's throughput before the camera path and light path can be connected. In the point light analogy, α can be interpreted as *radiant flux* [W]. In practice, α is usually multiplied with the BRDF of the last vertex in the light path (as an optimization under Assumption 3). As such, α stores *radiant intensity* $\left[\mathrm{W\,sr^{-1}}\right]$ in the point light analogy [Dachsbacher et al. 2014].

The latter is a finite element approach to solving the rendering equation.

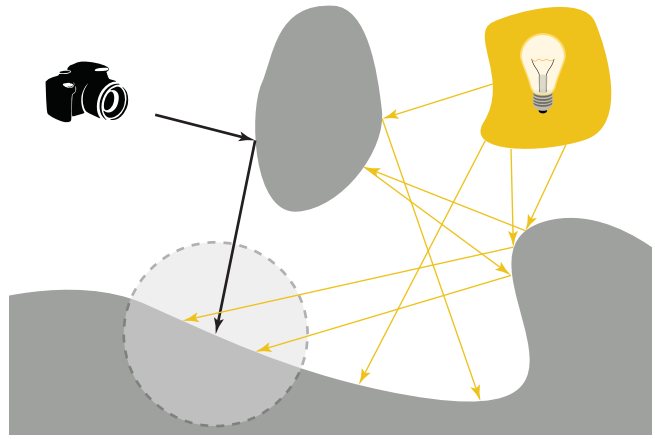


Figure 25: *Photon mapping. First, photons are traced from the light source into the scene. Second, a camera path is traced. At each intersection, the nearby photons are sampled.*

Rendering First, a camera path, DS^*E, is created (e.g., using Whitted ray-tracing). Then, the camera path is connected with the various light paths through the VPLs. This forms LD^+DS^*E paths. Consequently, the VPLs only contribute with indirect lighting (since an $L(D|S)E$ path is impossible). Therefore, direct lighting must be computed separately (e.g., using Whitted ray-tracing) and added to the result.

The camera path is connected with a light path by sampling the VPL that represents the light path. The VPL is sampled using

$$L_i(\mathrm{VPL}) = f_{\mathrm{VPL}}\alpha_{\mathrm{VPL}} \quad (15)$$

where the f_{VPL} term is the BRDF evaluated at x_{VPL}. As earlier mentioned, said term is multiplied directly onto the stored α_{VPL} in practice. Thus f_{VPL} is normally left out of Equation 15. All indirect lighting can be accumulated by connecting all light paths to the camera path. That is, as a sum of each VPL's contribution. Using this together with Equation 14 gives that

$$L_o(x,\omega_o,\omega_I) = \frac{1}{M}\sum_i^N f_r(x,\omega_o,\omega_i)L_i(\mathrm{VPL}_i)G(x,y) \quad (16)$$

where VPL_i is the i'th virtual point light out of N total. M is the number of generated light paths. Theoretically, each VPL's contribution should be weighted by the surface area represented by the corresponding light path vertex. This detail is often left out in practice. Instead, a global scale parameter is used.

Instant radiosity, as described above, is actually unbiased. However, sampling the same light paths leads to banding artifacts [Pharr and Humphreys 2004]. Furthermore, the G term is often bounded in practice to reduce so-called light splotches (due to singularities when $r \approx 0$). This bounding introduces bias in the algorithm [Dachsbacher et al. 2014].

Summary Instant radiosity is similar to bidirectional path tracing but has performance advantages. Specifically, that all light paths can be reused for each camera path. Many later methods are based on VPLs after the latter were introduced in instant radiosity [Dachsbacher et al. 2014]. Recently, VPL has also been used in real-time methods (Section 5.2). The real-time VPL methods have many similarities with our layered depth map approach which we will discuss later.

5.1.3 Photon Mapping

Photon mapping [Jensen and Christensen 1995], like instant radiosity, is a biased rendering method. First, the rendering equation must be rewritten in terms of *irradiance* $\left[\mathrm{W\,m^{-2}}\right]$. irradiance is differential radiant flux, $\mathrm{d}\Phi$, per differential area, $\mathrm{d}A$. It's denoted by

$$E(x,\omega) = \frac{\mathrm{d}\Phi(x,\omega)}{\mathrm{d}A}$$

Recall that radiance can be expressed in terms of irradiance

$$L(x,\omega) = \frac{\mathrm{d}^2\Phi(x,\omega)}{\mathrm{d}A\mathrm{d}\omega\cos\theta} = \frac{\mathrm{d}E(x,\omega)}{\mathrm{d}\omega\cos\theta} \tag{17}$$

where $\mathrm{d}\omega$ is the differential solid angle and θ is the angle between ω and the surface normal at x. Using Equation 17, the rendering equation can be written as an integral over irradiance

$$L_o(x,\omega_o) = L_e(x,\omega_o) + \int_{\mathcal{S}} f_s(x,\omega_o,\omega_i)\mathrm{d}E(x,\omega_i) \tag{18}$$

The key to photon mapping is to estimate the $\mathrm{d}E(x,\omega_i)$ term. This is done in two passes:

1. **Photon Tracing.** N photons carrying radiant flux are emitted from the light source and traced into the scene. Upon absorption, the photons are stored in a photon map.
2. **irradiance Estimation.** The photon map is queried in a local area around the point in question. The n nearest photons are used for the irradiance estimate.

Please refer to Figure 25 for an overview. The following paragraphs will provide more details about each step.

Photon Tracing Each photon, p, carries radiant flux

$$\Phi_p = \frac{\Phi_{\text{light}}}{n_e}$$

where Φ_{light} is the radiant flux of the light source and n_e is the total number of emitted photons. The photons are traced from the light source into the scene just like rays carrying radiance [Jarosz et al. 2008][12]. When a photon reaches a diffuse surface, the photon is either absorbed or reflected based on the outcome of Russian roulette. Upon absorption, the photon is stored in a photon map (a hierarchical data structure such as a k-d tree). Both the photon's radiant flux (Φ_p), position (x_p), and incoming direction (ω_p) are stored. Usually, two photon maps are used: A caustic photon map and a global photon map [Jensen and Christensen 1995]. The maps store LS^+D and $L(S|D)^*D$ photons, respectively. This division is done so that caustic photons (reflected or refracted photons) can be treated specially during rendering.

irradiance Estimation Recall that irradiance is radiant flux over area. The radiant flux is estimated, $E_{est}(x)$, as the N nearest photons to x. The original method is to search in a sphere centered at x that expands until the N nearest photons have been found [Jensen and Christensen 1995]. Let $r(x)$ be the final radius of the sphere. The irradiance estimate is then

$$E_{est}(x) = \frac{\Phi_N \text{ nearest photons}}{A_{\text{projected sphere}}} = \frac{\sum_{p=0}^{N}\Phi_p}{\pi r(x)^2} \tag{19}$$

[12]Refraction is an exception. Tracing of radiance must be scaled by the squared ratio of the medias' index of refraction, $\left(\frac{\eta_1}{\eta_2}\right)^2$, whereas radiant flux should not be scaled [Veach et al. 1996].

where Φ_p is the radiant flux of the pth nearest photon (out of N). The projected sphere is approximated by a circle so that $A_{\text{projected sphere}} = \pi r(x)^2$. Note how the estimate adapts based on the number of nearby photons. That is, if there are few photons near x then $r(x)$ will increase accordingly to cover a larger radius until the N nearest photons have been found. Likewise, if there are many photons near x then the N nearest photons will quickly be found and $r(x)$ will be small. Note that Equation 19 introduces bias since the radiant flux at x is averaged over multiple samples (which are not necessarily located in the immediate vicinity of x). The parameter N can be used to control the bias. Large N leads to systematic artifacts (blurred estimates) whereas low N leads to variance (noisy estimates).

This is an application of a broader method known as density estimation. The idea is to estimate an unknown function (E) based on data samples. The above method is a so-called adaptive density estimation method since the bandwidth, $r(x)$, is parametrized. Another well-known technique is the so-called kernel density estimation [Pharr and Humphreys 2004]. The sum in Equation 19 is generalized to associate a functional weight, w, to each sample

$$w(x) = \pi K\left(\frac{\|x - x_p\|}{r(x)}\right) \tag{20}$$

where $K(l)$ is the kernel function and x_p is the position of the pth photon. This way, photons closer to the point of interest (x) will weigh more in the sum. K is typically a smooth (continuous derivatives) function such that the density estimate itself will be smooth. In the case of photon mapping, the irradiance estimate is over a circle. A fitting K is usually normalized such that it integrates to $\frac{1}{\pi}$ (the inverse area of the unit circle). Using the kernel method leads to the following irradiance estimate

$$E_{est}(x) = \frac{1}{r(x)^2}\sum_{p=0}^{N}\Phi_p K\left(\frac{\|x - x_p\|}{r(x)}\right) \tag{21}$$

Note that the π has canceled out. Moreover, Equation 19 is a special case of Equation 21 when $K = \frac{1}{\pi}$. Another choice of K is Silverman's second-order kernel

$$K(l) = \begin{cases} \frac{\pi}{3}\left(1 - l^2\right)^2 & l < 1 \\ 0 & \text{Otherwise} \end{cases} \tag{22}$$

which has proven useful in practice [Frisvad et al. 2014].

Rendering The irradiance estimate (Equation 21) can be used to approximate the integral in Equation 18. This becomes the radiance estimate

$$L_o(x,\omega_o) = L_e(x,\omega_o) + \frac{1}{r(x)^2}\sum_{p=0}^{N} f_s(x,\omega_o,\omega_p)\Phi_p K\left(\frac{\|x - x_p\|}{r(x)}\right) \tag{23}$$

Note that x_p, ω_p, and Φ_p are all photon properties. Thus the incident radiance can be computed entirely from the glsirr estimate of nearby photons. In principle, this presents a unified (direct and indirect) lighting solution. In practice, however, computing all lighting using photon mapping is expensive. Instead, the incoming radiance is split into three parts: Direct ($L_{i,l}$), indirect specular ($L_{i,c}$), and indirect diffuse ($L_{i,d}$) [Jarosz et al. 2008]. $L_{i,l}$ is traced using a conventional method (e.g., Whitted ray-tracing). $L_{i,c}$ is computed using the sum in Equation 23 restricted to the caustic photon map. This preserves the high frequency details of the caustics. $L_{i,d}$ is

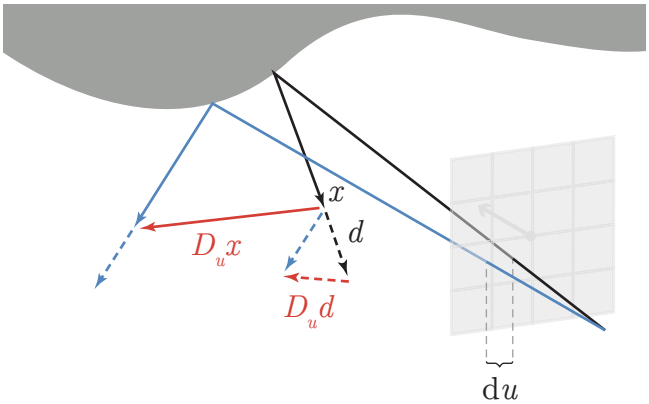


Figure 26: *Ray differential. The black ray is the main ray which we are currently tracing. The blue ray is the offset ray (which is not actually traced). The solid and dashed red vectors are the positional and directional ray differentials, respectively.*

computed using the sum in Equation 23 with the global photon map. Indirect diffuse lighting is low frequency so a blurred glsirr estimate is hardly noticed. The outgoing radiance is then the sum of each term

$$L_o = L_e + L_{i,l} + L_{i,c} + L_{i,d}$$

A single step of Monte Carlo integration can be used to improve the result of $L_{i,d}$ even further. This is called final gathering [Pharr and Humphreys 2004]. In this approach, the incoming diffuse radiance, $L_{i,d}$, is sampled using rays in random directions over the hemisphere. However, said rays use Equation 23 to sample $L_{i,d}$ at their first intersection. Thus the recursion stops immediately.

Summary Like VPLs, photon mapping has strong parallels to bidirectional path tracing. Similarly, it provides a unified approach to global illumination. The key difference is that photon mapping also handles specular reflections; a topic which is still actively researched for VPLs [Dachsbacher et al. 2014].

5.1.4 Photon Differentials

The main source of bias in photon mapping is the irradiance estimate. Furthermore, said estimate relies on the empirical parameter, N. Photon differentials is a method which improves on the irradiance estimate [Schjøth et al. 2007]. The classical tracing of a photon ray is augmented with a spread. That is, not only is the photon ray traced but also the differential spread of said ray. The differential spread forms a beam which is called the photon differential. Like the photon ray itself, the differential spread undergoes reflection and refraction. These interactions modifies the size of the photon differential (the footprint). In the end, the footprint is a measure of the photon differential's area. This can be used directly to compute an irradiance estimate.

Ray Differentials The theory behind photon differentials builds on ray differentials [Igehy 1999]. Though ray differentials were invented to improve texture filtering, much of the underlying theory remains the same. As such, we devote the next couple of paragraphs to the study of ray differentials.

Camera rays are typically computed using an image plane. Let $r = (x, d)$ be a ray where x is the ray's position and d is it's normalized direction. A camera ray corresponds to a set of uv-coordinates on the image plane[13]. The camera itself can be described by a projection model and a view model. The latter is generated from an eye point and a set of viewing directions: Forward, right, and up. The camera ray's initial position is simply the camera's eye point

$$x(u, v) = \text{Eye}$$

The camera ray's unnormalized direction, $\hat{d}$, is initially

$$\hat{d}(u, v) = t\text{Forward} + u\text{Right} + v\text{Up}$$

where u and v are in NDC and t is the distance to the ray's first intersection. It follows that the camera ray's normalized direction is simply

$$d(u, v) = \frac{\hat{d}}{\left\|\hat{d}\right\|} = \frac{\hat{d}}{\left(\hat{d} \cdot \hat{d}\right)^{\frac{1}{2}}}$$

Assume that two neighbouring camera rays were traced just slightly offset from the original ray's uv-coordinates

$$\begin{aligned} r_{\Delta u} &= r(u + \Delta u, v) = r(x(u + \Delta u, v), d(u + \Delta u, v)) \\ r_{\Delta v} &= r(u, v + \Delta v) = r(x(u, v + \Delta v), d(u, v + \Delta v)) \end{aligned}$$

for some small Δu and Δv. Together, the neighbouring rays form a ray beam. The positional difference between these rays is the beam's footprint. In turn, the footprint is a measure of the surface area represented by the 'pixel' corresponding to the ray beam. However, tracing two extra rays per pixel is inefficient.

The idea of [Igehy 1999] is to let $\Delta u, \Delta v \to 0$ and trace so-called ray differentials. As such, what is actually traced is the differential offsets themselves

$$\begin{aligned} D_u r &= (D_u x, D_u d) \\ D_v r &= (D_v x, D_v d) \end{aligned}$$

where D is the differential operator (Figure 26). In practice, the difference between the neighbouring rays is calculated using the first-order forward difference method

$$\begin{aligned} r(u + \Delta u, v) - r(u, v) &\approx \Delta u \cdot D_u r \\ r(u, v + \Delta v) - r(u, v) &\approx \Delta v \cdot D_v r \end{aligned}$$

for some Δu and Δv. In turn, this can be used to calculate the ray differentials footprint (we defer that to later). Note that only a single ray, r, is traced. The derivatives, $D_u r$ and $D_v r$, are updated accordingly using derivative tracing functions. The latter are the derivatives of the normal tracing functions. For instance, the ray differentials initial offsets, $D_u x$ and $D_u d$, are found by simply calculating the derivatives of the initial x and d functions

$$\begin{aligned} D_u x &= D_u \text{Eye} = \mathbf{0} \\ D_u d &= D_u \left(\frac{\hat{d}}{\left(\hat{d} \cdot \hat{d}\right)^{\frac{1}{2}}} \right) = \frac{\left(\hat{d} \cdot \hat{d}\right) \text{Right} - \left(\hat{d} \cdot \text{Right}\right) \hat{d}}{\left(\hat{d} \cdot \hat{d}\right)^{\frac{3}{2}}} \end{aligned}$$

where $\mathbf{0}$ is the zero vector. Analogously, an expression in the v-direction can be found.

Ray propagation (transferring) is another simple operation[14]. The purpose is to transfer a ray, $r = (x, d)$ from x in direction d to

[13] We use uv in NDC to avoid confusion with xy in WC.

[14] Assuming that it happens in a homogenous medium.

x^*. Let t be the distance to the ray's first intersection. Then the transferred ray, $r^* = (x^*, d^*)$, is

$$\begin{aligned} x^* &= x + td \\ d^* &= d \end{aligned}$$

This is elemental ray-tracing. Note that the direction doesn't change; reflection and refraction are dealt with in another step. The ray differential equivalents are more involved. Taking the derivative of the above leads to

$$\begin{aligned} D_u x^* &= (D_u x + t D_u d) + (D_u t)\, d \\ D_u d^* &= d \end{aligned}$$

To find $D_u t$, an expression for t is first needed. Let N be the surface normal at x^*. Assume that the surface at x^* is a plane tangent to N. Furthermore, assume that x is given in a coordinate system centered at x^*. Using similar triangles, t can be calculated as

$$t = -\frac{x \cdot N}{d \cdot N}$$

Now the derivative of t can be readily found

$$D_u t = -\frac{(D_u x + t D_u d) \cdot N}{d \cdot N}$$

using that $D_u N = N$ (due to the tangent plane assumption). Note that the coordinate system assumption has no impact on the derivative since no absolute positions are involved. Likewise, it can be shown that the tangent plane assumption isn't necessary [Igehy 1999]. Thus the above equation for $D_u t$ holds in general. Analogously, an expression for v can be derived.

Similar derivations can be used to find the derivative functions for reflection and refraction. These functions are not needed in our case so we omit them here. Please refer to [Igehy 1999] for a full treatment on ray differentials.

Photon Connection Ray differentials are directly connected to photon differentials through emission from a point light source [Schjøth et al. 2007]. A spot light, for instance, can be modelled as a pinhole camera. As such, each pixel on the image plane corresponds to a photon emission. The differential of said photon can then be traced directly as one would trace ray differentials. In this context, the photon's position, x_p, becomes the ray position and similarly, $\omega_p = -d$.

Thus it is now possible to trace radiant flux simultaneously with a measure of the photon's footprint. The photon's positional differential is the key. Said differential spans a parallelogram with area

$$A_r = |D_u x_p \times D_v x_p|$$

where A_r is the area of the ray footprint. The area of the photon footprint, A_p, is the max-area ellipse inscribed in the parallelogram [Frisvad 2012]

$$A_p = \frac{\pi}{4} A_r = \frac{\pi}{4} |D_u x_p \times D_v x_p|$$

Using this, the irradiance estimate for a single photon, E_p, can be found directly as

$$E_p = \frac{\Phi_p}{A_p} \tag{24}$$

Like in photon mapping, the irradiance can be directly used to estimate the reflected radiance in Equation 18. The outgoing radiance becomes

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \sum_{p=0}^{N} f_s(x, \omega_o, \omega_p) E_p \tag{25}$$

where the sum is over the N photons whose footprint overlaps x.

Kernel Method Like in photon mapping, a kernel function, K, can be used to weigh the photon's contribution based on it's distance to x [Schjøth et al. 2007; Frisvad 2012]. However, simply using $\|x - x_p\|$ directly is naive since it does not take the photon's (possibly anisotropic) footprint into consideration. The problem is to find a matrix, M_p, mapping from WC into *filter coordinates* (FC) the latter being the coordinate space spanned by the photon differential (an ellipsoid). Fortunately, mapping the other way around is straightforward. It can be expressed in terms of a matrix, B_p, using the positional differentials and the surface normal

$$B_p = \begin{bmatrix} \frac{1}{2} D_u x_p \\ \frac{1}{2} D_v x_p \\ n_p \end{bmatrix}$$

where n_p is the surface normal at x_p. Half the length of the positional differential vectors are used to center around x_p. Thus $M_p = B_p^{-1}$. Inverting a 3×3 matrix can be done with a few cross products [Collomb 2007] so that

$$\begin{aligned} M_p &= \frac{1}{\left(\frac{1}{2} D_u x_p\right) \cdot \left(\left(\frac{1}{2} D_v x_p\right) \times n_p\right)} \begin{bmatrix} \left(\frac{1}{2} D_v x_p\right) \times n_p \\ n_p \times \left(\frac{1}{2} D_u x_p\right) \\ \left(\frac{1}{2} D_u x_p\right) \times \left(\frac{1}{2} D_v x_p\right) \end{bmatrix} \\ &= \frac{2}{D_u x_p \cdot (D_v x_p \times n_p)} \begin{bmatrix} D_v x_p \times n_p \\ n_p \times D_u x_p \\ a n_p \end{bmatrix} \end{aligned} \tag{26}$$

where a should in principle be $\frac{1}{2}$. However, a can instead be interpreted as a parameter which controls topological bias due to differences in normal orientation. Alternatively, the last row, $a n_p$, can be omitted for added performance [Frisvad et al. 2014].

Equation 25 is then modified to include kernel weighing

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \sum_{p=0}^{N} f_s(x, \omega_o, \omega_p) E_p \pi K(\|M_p(x - x_p)\|) \tag{27}$$

The kernel function, K, can be any of the previously-mentioned functions. The sum is still over the photons whose footprint overlap with x.

Scaling Note that the size of the photon footprint directly controls the bandwidth of the radiance estimate. A scaling parameter, s, is introduced which globally scales every photon differential. This way, s can be used to control the bandwidth of the radiance estimate [Frisvad et al. 2014]. As such, large s increases the bias (blurring the result) whereas small s reduces bias but introduces variance (noise in the result). Practically, s is used as an empiric parameter to scale the photon footprints so that the latter cover the whole scene. Note that energy is conserved since the footprint directly corresponds to the area in the irradiance estimate so that the photon's radiant flux is spread out accordingly.

Splatting Conventionally, the position, x, would be used to index into a map of photon differentials to find the overlapping footprints. The sum in Equation 27 can then be computed directly. This is the standard ray-tracing approach (pixel→photon differentials). Another approach is to splat the photon differential directly onto the image plane (photon differential→pixels) [Frisvad et al. 2014]. This has the added benefit that no photon map needs to be stored and thus no costly lookups into said map.

With a rasterization-based pipeline, primitive→pixels is the natural order. Therefore splatting is an ideal approach in our use case. We will go into further details when we design our method.

5.2 Previous Work

This section describes how the indirect lighting methods mentioned in Section 5.1 can be implemented. The discussion is focused on approaches which are either based layered depth maps or relevant in our comparison. Still, we will also mention other methods to establish historical context. We go into further details with methods which are also real-time and used with rasterization. In a later section, we will compare our indirect lighting approach to the previous work described in this section.

5.2.1 Reflective Shadow Maps

A conventional shadow map is a depth map generated from the light's view. A reflective shadow map augments the shadow map by also storing radiant flux, surface normal, and surface position in WC [Dachsbacher and Stamminger 2005]. As such, each pixel in the reflective shadow map is a VPL representing an LD path. Reflected radiance from the camera's view is then integrated in screen-space. That is, nearby pixels (nearby VPLs) are sampled in a fragment shader and the result is accumulated.

The problem with this approach is that the visibility term, V, is not evaluated. This is for performance reasons since creating a shadow map for each pixel in the reflective shadow map is impractical. Instead, it is proposed to use AO to compensate for the missing shadowing [Dachsbacher and Stamminger 2005]. Still, light leaks can occur between surfaces that are actually mutually occluded.

Moreover, reflective shadow maps are limited to a single bounce of light. This is a limitation of using a rasterization-based approach since DE paths are rendered. As such, only an $LDDE$ path can be formed using reflective shadow maps.

5.2.2 Imperfect Shadow Maps

As previously mentioned, imperfect shadow maps are coarse approximations of shadow maps [Ritschel et al. 2008]. As such, imperfect shadow maps can be generated much faster. Imperfect shadow maps can be combined with reflective shadow maps to compute the V term of each VPL. This removes the light leak problem of reflective shadow maps. Note that imperfect shadow maps can also be combined with any other VPL-based approach. It is just natural to consider reflective shadow maps since this approach also works in a rasterization-based pipeline.

An imperfect shadow map is generated by sampling a point on each primitive (instead of the full primitive). Said point is then rasterized as a small screen-aligned quad. This is indeed a coarse approximation but it works well in practice. Furthermore, a so-called push-pull approach can be used to fill in gaps between neighbouring points [Ritschel et al. 2008]. A single rasterization pass can generate multiple imperfect shadow maps simultaneously by splitting the incoming points equally (and randomly) between the imperfect shadow maps. As such, each imperfect shadow map will only contain information about a small subset of the scene. Using the push-pull method, however, holes in this subset can be coarsely filled.

Imperfect reflective shadow maps [Ritschel et al. 2008] are the imperfect shadow map analogues to reflective shadow maps. By augmenting each of the imperfect shadow maps with light information, multiple light bounces of indirect light can be computed. Of course, this adds additional complexity to the method and thus degrades performance.

Being imperfect, the V term is sometimes not approximated correctly. Consequently, light leaks may still occur. Increasing the resolution of each reflective shadow map can improve the approximation at the cost of performance. The optimal resolution must be found empirically.

5.2.3 Parallel Global Ray-bundles

A global ray-bundle is a set of parallel rays. Along each ray, the scene intersections are recorded. Hopefully, this should sound familiar as a layered depth map can be interpreted as a set of rays intersection the scene (Section 3.3). Global ray bundles can be used in a combined Monte Carlo and finite element method to compute indirect lighting [Sbert and Sàndez 1996; Hermes et al. 2010].

The underlying idea is that two consecutive depth values in an L_p sequence form a DD path (when the corresponding surfaces are connected through open space). As such, the layered depth map can be used to transfer radiance in both directions of the DD path. This is done for all such DD paths in the layered depth map. Moreover, the process is repeated for multiple layered depth maps each oriented in a different direction sampled uniformly at random. This is similar to path tracing but with coherent rays and always terminating after the first bounce. The outgoing radiance values are retrieved from a texture atlas, TA_o, which has an entry for each surface point. Likewise, the incoming radiance is stored in an equivalent texture atlas, TA_i. After all radiance transfers, the two atlases, TA_o and TA_i, are swapped and another round of radiance transfer is initiated. For each round, another bounce of indirect lighting is computed[15]. Thus the method can compute an arbitrary amount of light bounces.

The algorithm is specifically interesting since the authors propose to use a k-buffer to generate the layered depth maps [Hermes et al. 2010]. They do not claim it to be real-time, however. Still, this approach was an inspiration for other real-time global illumination methods such as our own. Section 5.2.4 describes another similar method.

5.2.4 VPL-based Hybrid

A hybrid VPL and path tracing approach can be implemented using layered depth maps [Tokuyoshi and Ogaki 2012b]. The idea is to combine the two approaches into a single bidirectional algorithm. To do so, a reflective shadow map is first used to generate the VPLs along with a regular shadow map for each VPL. As explained earlier, this forms LD paths. Then, global ray-bundles are generated using layered depth maps[16]. Recall that two consecutive depth values in a L_p sequence form a DD path. Lastly, DE paths are rendered from the camera into a G-buffer.

Any combination of the above-mentioned paths can be connected. E.g., an $LDDDE$ path by tracing from the eye to the first surface (using the G-buffer), then reflecting towards another surface (using a layered depth maps), and lastly towards a VPL (using the corresponding shadow map for visibility). This results in a two-bounce global illumination method. Likewise, single-bounce paths, $LDDE$, can also be found by omitting either the DD step or sampling the light source directly (and not the VPL). Furthermore, additional bounces can be added by tracing layered depth maps for more DD paths. That is, LD^*E paths are possible though at the cost of performance for each additional bounce.

This method is specifically worth mentioning, since it is suggested to use PPSLLs to implement the layered depth maps [Tokuyoshi and Ogaki 2012b]. Moreover, because the layered depth maps is

[15] This is similar to the finite element radiosity method [Goral et al. 1984].

[16] The method of [Tokuyoshi and Ogaki 2012b] is inspired by [Hermes et al. 2010]. Therefore, the authors use the term global ray-bundles.

used to trace parallel rays. This method proves that layered depth maps can be used to implement indirect lighting in real-time with a rasterization-based pipeline.

The authors propose to use unsorted depth value sequences to reduce the layered depth maps's construction time. As such, the DD paths must be found using a linear search which always goes to the end of the list. Recall that using PSPPSLLs, this linear search can be terminated early. Though note that PSPPSLLs were not documented in the literature at the time the VPL-based hybrid was presented.

Imperfect Ray-bundle Tracing The principle behind imperfect shadow maps can also be applied to Ray-bundle tracing [Tokuyoshi and Ogaki 2012a]. In this approach, the layered depth maps are generated from a point-based representation of the scene. That is, each primitive in the scene is represented by a point instead. Unlike imperfect shadow mapping, each point is then rasterized as a circle and not quads. The latter proved to be too rough an approximation for this use case. The circles are then used in layered depth map construction. The result is that fewer list nodes are generated resulting in much faster rendering times. Of course, light leaks can now occur since the point-based scene representation is coarse.

Predecessors The idea to rasterization to generate a coherent set of parallel rays is old [Hachisuka 2005]. The problem has historically been to generate the layered depth maps fast enough for real-time purposes. For instance, the depth peeling approach is suggested in [Hachisuka 2005]. Recall that we discarded depth peeling early on since it requires a full geometry pass for each layer in the L_p sequence (and discards each layer instead of storing them).

The idea to use coherent rays to accelerate ray-tracing to interactive rates is even older [Wald et al. 2002]. The idea to just use coherent rays traces back to the so-called global line radiosity method [Dutré et al. 2006]. A similar line-based approach is known as intersection fields [Ren et al. 2005]. However, these methods are not directly based on layered depth maps so we will not go into further details with them.

Offline Derivatives The method of [Hermes et al. 2010] can also be implemented using PPSLLs to increase performance [Tokuyoshi et al. 2011]. Though it is still intended for offline use (to compute light maps). An interesting memory optimization is to use a tiled multi-pass approach [Tokuyoshi et al. 2013]. In the latter, the PPSLLs are generated only for a small tile of the framebuffer at a time (one tile in each pass). The memory usage is vastly reduced since far fewer list nodes have to be stored for each pass. Of course, this assumes that the unbounded memory requirements of PPSLLs are a problem. This may indeed be the case in offline rendering where the scenes are typically for more complex than in real-time rendering.

5.2.5 Ray-marching Layered Depth Maps

In the aforementioned methods, the layered depth map has been used to trace rays in the direction which the layered depth map is oriented. This requires the construction of a layered depth map for each direction. Another approach is to trace in arbitrary directions by ray-marching through the layered depth map [Lischinski et al. 1998; Bürger et al. 2007]. This approach is analogous to ray-marching a depth map (as done in HBAO). Instead of testing a single depth values per pixel, the L_p sequences must be searched through. As expected, this is a slow process. On the other hand, fewer layered depth maps needs to be generated. In fact, only three layered depth maps in orthogonal directions are necessary; a so-called layered depth cube [Lischinski et al. 1998]. When tracing in direction, ω, the layered depth map which is oriented closest to ω is chosen. Then the chosen layered depth map is ray-marched.

Three orthogonal layered depth maps is the theoretical minimum. In practice, rays that are almost orthogonal to the chosen layered depth map will have to sample many L_p sequences. This can be mitigated by using additional layered depth maps. The more layered depth maps, the less so-called pixel crossings (and the less L_p sequences will have to be sampled) [Niessner et al. 2010]. As such, it is sometimes more performant to use more than the theoretical minimum number of layered depth maps in practice . The optimal amount depends on the underlying implementation.

The ray-marching scheme is not perfect. Rays may miss intersections for steep depth values. A tolerance threshold for the intersection test can mitigate this issue [Niessner et al. 2010].

Applications include: Whitted ray-tracing of reflections [Bürger et al. 2007], glossy reflections and soft shadows [Niessner et al. 2010], caustic photon tracing [Krüger et al. 2006], and path tracing [Hu et al. 2014]. The first two are offline methods, the third is interactive, and the last is real-time. We will go into further details with the last method in the following paragraphs. Note the variety of global illumination methods that have been implemented with the ray-marching approach. Since directions can be chosen freely, only performance restricts these approaches.

Voxel Grid Hybrid As briefly mentioned in Section 2, coarse scene representations via voxel grids can be constructed in real-time. A hybrid approach uses voxel grids for coarse ray-scene intersections and then refines the intersection by ray-marching using three orthogonal layered depth maps [Hu et al. 2014]. I.e., first an intersection interval is found by tracing through the voxel grid in the given direction, ω. Then, the layered depth map closest to ω is ray-marched to find the precise intersection. Note that the intersection interval from the voxel grid can be used to narrow the ray-marching to only a few L_p sequences. Thus the problem mentioned earlier with using the theoretical minimum number of layered depth maps is effectively mitigated.

The voxel grid is simply a uniform grid. The layered depth maps are implemented using PPSLLs. As such, a linear search (without early termination) is needed to find the relevant depth value. Also, while the coarse voxel intersection interval may narrow the ray-marching, multiple L_p sequences may still have to be searched in full. Still, the method produces convincing results in real-time. The authors also propose to use progressive path tracing (and only reconstruct the voxel grid and layered depth maps when the scene changes) [Hu et al. 2014].

5.2.6 Photon Differentials

Section 5.1.4 skipped how photon differentials can be reflected or refracted. Specular reflections and refractions can be described by one to one mappings of ingoing and outgoing directions. As such, the derivative specular reflection and refraction functions can be readily found [Igehy 1999]. Thus it is possible to use photon differentials for caustics [Frisvad et al. 2014]. Diffuse reflections, however, do not have a simple mapping of incoming to outgoing directions. This makes it difficult to find an expression for the derivative tracing function.

Path Differentials Ray differentials have later been generalized to so-called path differentials [Suykens and Willems 2009]. Path differentials generalize the ray differentials to other domains besides the image plane (uv-coordinates). Specifically, differentials based on random sampling can be calculated. For every tracing event

(transfer, reflection, refraction, etc.), the path differential stores the differential parameters used in the generation of the new path vertex. Such a generalized parametrization can also be used to compute photon differentials from arbitrary light sources (generalizing from just point lights) [Frisvad et al. 2014].

Diffuse Reflection Path differentials introduce the mechanism needed for diffuse reflection: Sampling the outgoing direction at random (as done in path tracing). However, the number of parameters in the path differential grows for every interaction which is unwanted. The solution of [Fabianowski and Dingliana 2009] is to interpret diffuse reflection as an absorption directly followed by a re-emission. This ensures the set of differential parameters is constant which is useful for interactive purposes. Specifically, the photon differential is reconstructed as if the photon had just been emitted from the light source (but now in the diffusely reflected direction). With re-emission, however, all previous scene interactions recorded in the photon differential is lost. The solution is to virtually offset the photon before re-emission. The virtual offset is based on the current positional and directional differentials (see [Fabianowski and Dingliana 2009] for the specifics). Thus previous scene interactions are roughly retained in the differential.

Russian roulette can also be applied to path differentials [Suykens and Willems 2009]. Again, such a stochastic mechanism requires additional parameters to be stored in the differential. To keep the number of parameters constant, the existing differentials can instead be scaled according to the outcome of the Russian roulette [Fabianowski and Dingliana 2009]. That is, by increasing the size of the photon footprint by a factor $\frac{1}{p}$ where p is the probability of the event occurring. As in conventional photon mapping, Russian roulette is used to terminate the tracing stochastically.

Using the above-mentioned methods, diffuse reflections are possible and can be done at interactive rates [Fabianowski and Dingliana 2009]. The authors use GPU acceleration through CUDA. Equation 27 is computed conventionally by using a bounding volume hierarchy as the photon map.

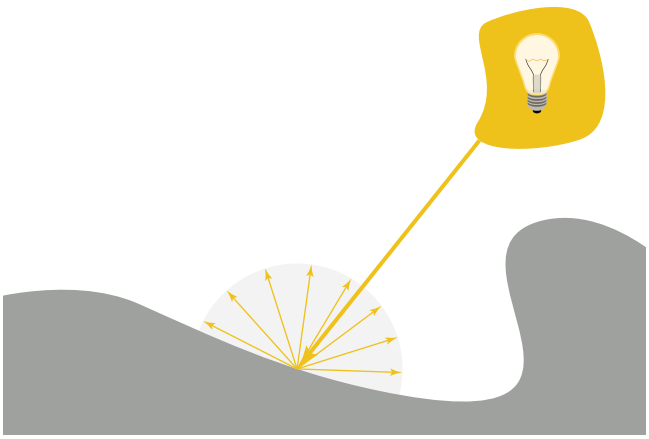


Figure 27: *Photon splitting. The primary photon is split into multiple secondary photons. Each secondary photon is weighed according to the BRDF at the surface intersection.*

5.3 Design

We will now describe our approach to indirect lighting using on layered depth maps. As mentioned earlier, we base our method on photon differentials. The layered depth maps are used for photon tracing as inspired by previous approaches. First, we outline a method to diffusely reflect photon differentials. Second, we outline an algorithm which can be used to implement photon differentials in a rasterization-based pipeline. Third, we propose a slight modification which allows us to omit the photon storage.

5.3.1 Deterministic Diffuse Reflection of Photon Differentials

Traditionally, photons are traced as atomic quantities. That is, photons are either reflected or absorbed but never split in two [Jarosz et al. 2008]. Russian roulette is used to determine which event will occur (and the photon's radiant flux is weighed accordingly). This ensures two things:

- The total number of photons is kept constant. I.e., if n_e photons are emitted from the light sources then n_e will be stored in the photon map.
- Photons are prioritized according to the number of light bounces because each bounce depends on the outcome of the former. E.g., the fourth bounce will not even occur of the third bounce was absorption. Thus longer photon paths are less likely than shorter ones.

The first property ensures that the memory requirements for the photon map are bounded. The second property is good because light becomes less visually important the more it bounces around. That is, the first few light bounces are enough to create visually convincing results.

Alternatively, reflection can be modelled by splitting photons [Jarosz et al. 2008]. A single photon is split into multiple photons each weighed by the surface properties at the point of reflection (Figure 27). E.g., a photon can be split into two new photons: One for the diffuse direction and another for the specular direction (and weighed accordingly). The problem with such an approach is that the number of photons increases exponentially with the number of light bounces. On the other hand, photon splitting is deterministic. Still, diffuse reflection is a problem since there is not a one-to-one mapping between the incoming and outgoing direction (as in a perfect specular reflection). We propose to choose the outgoing directions deterministically. Specifically, we sample the hemisphere uniformly using the deterministic approach described in Section 4.3.2. This results in N fixed outgoing directions. Thus we propose to split the incoming photon into N outgoing photons; one in each of the outgoing directions. To limit the number of generated photons, the photons are absorbed after the first bounce. This our method produces $LDDE$ paths. See Section 7.2.1 for a multi-bounce extension. Each of the N outgoing photons are weighed by the surface's BRDF. Since the N outgoing directions are sampled uniformly, this approach fits best to Lambertian surfaces with a constant BRDF. However, it is not limited to such and any BRDF can be used (though many unimportant directions may be traced in vein).

Updating the Photon Differentials To find a derivative diffuse reflection function, we first describe regular diffuse reflection in terms of the above approach. Let ω_i be the incoming direction of the photon and let ω_o be one of the N outgoing directions in which a new photon is traced. Upon diffuse reflection, the ray $r = (x, d)$ results in the $r^* = (x^*, d^*)$ where

$$\begin{aligned} x^* &= x \\ d^* &= \alpha(\omega_i, \omega_o) \cdot d \cdot \bar{\alpha}(\omega_i, \omega_o) \end{aligned}$$

The $\alpha(\omega_i, \omega_o)$ term is the rotation quaternion which represents the rotation of vector ω_i to vector ω_o. $\bar{\alpha}$ is the conjugate of α. Note that d is implicitly converted to a pure quaternion (and back). The $\cdot$ operator is the Hamilton product. Informally, the expression $qd\bar{q}$ denotes the rotation of d by the rotation quaternion q. We will describe quaternion rotation in more detail shortly. Note that α does not depend on neither x or d. Thus α is also independent of the corresponding uv-coordinates. This leads to the following straightforward derivative diffuse reflection functions

$$\begin{aligned} D_u x^* &= D_u x \\ D_u d^* &= \alpha(\omega_i, \omega_o) \cdot D_u d \cdot \bar{\alpha}(\omega_i, \omega_o) \end{aligned}$$

The positional differential is unchanged and the directional differential is rotated according to α. Analogous expressions can be derived for the v-coordinate.

Defining α The α function is described in full detail in [Sam 2014]. We will repeat the important parts here. A rotation quaternion, q, is defined as

$$\begin{aligned} q(v, \theta) &= \left(\sin\frac{\theta}{2} v_x, \sin\frac{\theta}{2} v_y, \sin\frac{\theta}{2} v_z, \cos\frac{\theta}{2} \right) \\ &= \sin\frac{\theta}{2} \left(i v_x + j v_y + k v_z \right) + \cos\frac{\theta}{2} \end{aligned}$$

where v is the rotation axis (a unit vector) and θ is the angle of rotation. i, j, k are the unit vectors spanning the Cartesian coordinate system. Recall that $\alpha(\omega_i, \omega_o)$ is the rotation quaternion which represents the rotation of vector ω_i to vector ω_o. The function $\alpha(\omega_i, \omega_o)$ can then be defined as follows

$$\alpha(\omega_i, \omega_o) = q\left(\frac{\omega_i \times \omega_o}{\|\omega_i \times \omega_o\|}, \cos^{-1}(\omega_i \cdot \omega_o) \right)$$

assuming that ω_i and ω_o are normalized. The $\times$ operator is the vector cross product and $\cdot$ operator is the vector dot product. α can be understood intuitively by inspecting the geometric operations. The cross product produces a vector orthogonal to both operands. This

becomes the rotation axis (v). The dot product results in the cosine of the angle between the two vectors. This becomes the rotation angle (θ).

By further assuming that ω_i and ω_o are not parallel, it can be shown [Sam 2014] that α reduces to

$$\alpha(\omega_i, \omega_o) = \frac{(\alpha_x, \alpha_y \alpha_z \alpha_w)}{\|(\alpha_x, \alpha_y \alpha_z \alpha_w)\|}$$

where

$$\begin{aligned} (a_x, a_y, a_z) &= \omega_i \times \omega_o \\ \alpha_w &= 1 + \omega_i \cdot \omega_o \end{aligned}$$

The $\|\cdot\|$ operator is the Euclidean norm. This approach readily applies to graphics hardware.

Quaternion Rotation Let d be the vector that should be rotated by rotation quaternion, q. The resulting vector, d^* is then

$$d^* = q \cdot d \cdot \bar{q}$$

where $\bar{q}$ is the conjugate of q and the $\cdot$ operator is the Hamilton product [Baker 2015]. Note that d is implicitly converted to a pure quaternion (the coordinate is zero, $w = 0$) and back again (by discarding the w-coordinate). Conjugation is simply $\bar{q} = (q_x, -q_y - q_z - q_w)$. The Hamilton product is more involved. It can be shown [JeGX 2014] that the rotation reduces to

$$d^* = d + 2\left(q_{xyz} \times (q_{xyz} \times d + q_w v)\right)$$

which readily applies to graphics hardware.

5.3.2 Photon Differentials with Layered Depth Maps

The choice of deterministic uniform sample directions directly maps to tracing in layered depth maps. As such, we can also use layered depth maps as an auxiliary data structure for tracing photon differentials. Furthermore, the operations used to update the photon differentials readily applies to graphics hardware. We propose an algorithm which requires two passes (besides layered depth maps construction):

1. **Photon Tracing.** Photon differentials are traced from a light source into the scene. Upon the first intersection, the incoming photon differential is split into N outgoing photon differentials according to the BRDF at the intersected surface. Said photon differentials are then stored in an unordered photon buffer.

2. **Photon Splatting.** The photon differentials from the photon buffer is splatted onto the image plane. The extent of the splats are determined from the photons footprint.

Note that we propose to use an unordered photon buffer. We use this notation since photon maps typically implies a hierarchical structure. In contrast, our photon buffer is simply flat and unsorted. The photon buffer can be implemented using a simple array of contiguous memory. We will go into further detail in the following paragraphs

Photon Tracing We limit our methods to point light sources. As such, the photon tracing can be initiated by rendering the scene from the light's point of view. This has the added benefit that we can use the ray differential theory (explained earlier) to trace the photon differentials with respect to the uv-coordinates. A photon is emitted for each pixel rendered. Thus the resolution of this step determines how many primary photons are emitted. We denote this resolution R_{light}. The tracing itself is done in a fragment shader. The next point of intersection in the photon's outgoing direction is found with trace(r) function (Section 3.3) via layered depth maps. A total of N secondary photons are emitted for each primary photon (each in their own direction). Finally, the photon differentials are stored in an SSBO in arbitrary order. Specifically, we store the photon's position in WC, the normal of the intersected surface, the positional differential, and the photons radiant flux. The latter two are used in the irradiance estimate.

Photon Splatting The photon buffer is rasterized as a set of points. A geometry shader expands each point into a quad aligned with the photon's footprint. Lastly, a fragment shader splats the photon's contribution to the framebuffer. This is similar to the work of [Frisvad et al. 2014] but using rasterization instead of searching for eye paths that overlap with the footprint. Thus we evaluate each term of the sum in Equation 27 independently and splat the result to the relevant pixels. The result is the same. We use Silverman's second-order kernel (Equation 22) for K as done in [Frisvad et al. 2014].

5.3.3 Skipping the Photon Buffer

In principle, it is possible to combine the **Photon Tracing** and **Photon Splatting** passes into a single pass. That is, by splatting the photon directly as soon as it has been traced. This saves both the overhead of the second pass and storing the photon buffer. The problem is that the splatting cannot be done directly to the framebuffer since the latter is currently mapped to the tracing algorithm. Thus we need a mechanism that allows a fragment shader to write to an arbitrarily large region of an auxiliary image. This can be done with the the so-called image-load-store extension [Bolz et al. 2014b] or using an SSBO.

The next problem is to rasterize the photon splat directly in the fragment shader. This can quickly become involved. We propose to use the photon splat's screen-aligned quad is this can be trivially found from the positional differential. The problem with using a screen-aligned quad is that a lot of computational power may be wasted on pixels where the splat doesn't contribute. Still, this is mostly a problem for splats which are diagonal in image-space. For splats which are coarsely screen-aligned, the overhead is negligible.

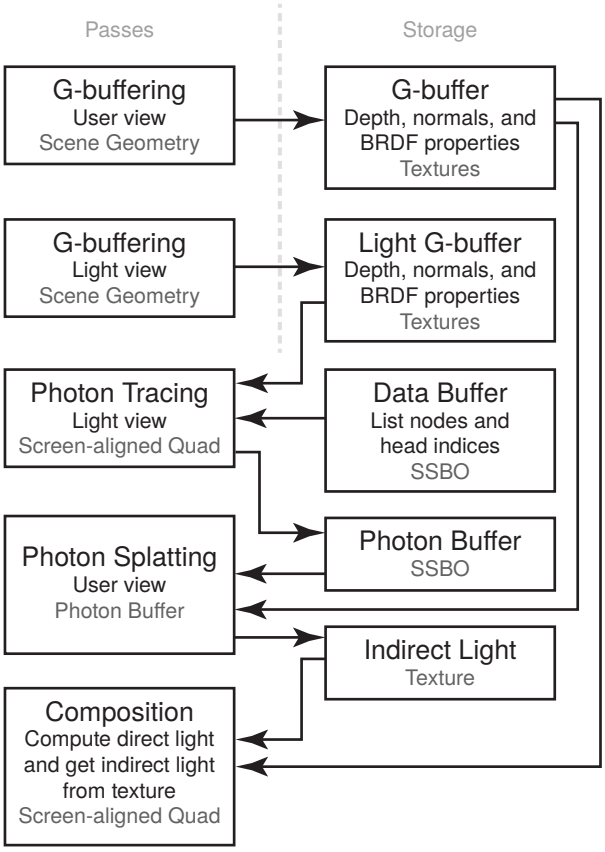


Figure 28: *Overview of photon tracing and splatting.*

5.4 Implementation

In this section, we will describe how to implement our indirect lighting method. First, we describe how the tracing pass is implemented. Second, the splatting pass is detailed. Third, we explain why the photon buffer is actually necessary in practice.

5.4.1 Tracing Photon Differentials Using Layered Depth Maps

This pass is rendered over the scene geometry from each lights' point of view. It is described in the context of a fragment shader. The corresponding vertex shader is trivial and has been omitted. The tracing pass is divided into the following steps

1. **Calculate Radiant Flux.** Each photon's radiant flux, Φ_p, is based on the light's total radiant flux, Φ_{light}.

2. **Initialize Photon Differential.** This is done using the pixel's uv-coordinates and the light's orientation. Let x_0 be the photon's position on the light source and let d_0 be it's initial direction.

3. **Transfer Photon Differential.** Transfer from x_0 to the first intersected surface, x_1. The direction is unchanged, so $d_1 = d_0$.

4. **Split Photon.** Let N be the number of layered depth maps. Then a photon is traced in both directions of each layered depth map (totaling in $2N$ photons). For each corresponding direction d_2:

   (a) Compute $x_2 = \text{trace}(x_1, d_2)$; the intersection with the first diffuse surface.

   (b) Project x_2 into the user's view. Discard the photon if it is not visible.

   (c) Diffusely reflect the photon differential from d_1 to d_2.

   (d) Transfer the photon differential from x_1 to x_2.

   (e) Store the photon differential in the photon buffer.

In the **Split Photon** step, the photon is first traced (a) before the differential is updated (c,d). This is done is done so that occluded photons can be rejected early (b). The following paragraphs will go into details. Also note that the primary photons (from the light source) are not stored. Only indirect photons are stored. As such, the splats will only contribute with indirect lighting. A third pass is needed to compute direct lighting. This can be done using conventional rasterization. Please refer to Figure 28 for an overview.

Calculate Radiant Flux The light's total radiant flux, Φ_{light}, must be split between all the photons. Recall that the formula is

$$\Phi_p = \frac{\Phi_{\text{light}}}{n_e}$$

where n_e is the number of emitted photons. We split this into two quantities: The number of primary photons ($n_{e,p}$) and secondary photons ($n_{e,s}$). The former originates from the light source and the latter are the split photons. A primary photon is emitted for each fragment. Thus $n_{e,p}$ is a function of the resolution, R_{light}. To produce a spot light, however, fragments outside the unit circle (in half *texture coordinates* (TC)) are culled

```
// Radius in half texture coordinates [0;0.5]
float radius = length(gl_FragCoord.xy /
    window_dimensions - vec2(0.5));
// Discard fragments outside the unit circle
if (radius > 0.5) discard;
```

Note that we don't normalize to $[0; 1]$ since it is redundant[17]. Thus we must account for the fragments lost due to this culling. The ratio of unculled fragments to the initial amount of fragments is

$$\frac{A_{\text{unit circle}}}{A_{\text{unit square}}} = \frac{\pi r^2}{(2r)^2} = \frac{\pi}{4}$$

Thus the total number of primary photons is $n_{e,p} = \frac{\pi}{4} R_{\text{light}}$.

Next, the number of secondary photons must be estimated. For each primary photon, $2N$ secondary photons are emitted. Thus $n_{e,s} = 2N n_{e,p}$. Since only photons due to indirect lighting are stored, we use

$$n_e = n_{e,s} = \frac{\pi}{2} N R_{\text{light}}$$

The corresponding GLSL code is

```
// The light's radiant flux is found in a uniform
const vec4 Phi_light = /* */;
const int R_light = light_view_dimensions.x *
    light_view_dimensions.y;
```

[17] This step can be further optimized by computing the squared length with a dot product instead. We simply use the `length` function for pedagogical reasons.

```
const int n_e = int(2.0 / PI * N * R_light);
// Each photon's radiant flux
const vec4 Phi_p = Phi_light / float(photon_count);
```

Initialize Photon Differential We directly apply the formulas given in Sections 5.1.4. We use the following self-explanatory structure for organization

```
struct photon_differential
{ vec3 Du_x, Dv_x, Du_d, Dv_d; };
```

The construction routine itself is straightforward but lengthy. It is given in Listing 7. We call it like this

```
// Rename for consistency with theory
vec3 d_hat = vertex.wc_view_ray_direction;
photon_differential photon =
    construct_photon_differential(d_hat,
    current_view.right, current_view.up);
```

where the `vertex.wc_view_ray_direction` is passed in directly from the vertex shader. This is part of a trick to cheaply reconstruct the position in WC in rasterization [Mittring 2007].

Transfer Photon Differential The transfer function is likewise straightforward but lengthy. It is given in Listing 8. It is called as follows

```
// Rename for consistency with theory
vec3 wc_n = wc_normal;
vec3 d = normalize(d_hat);
float t = -ec_position_z;
transfer(ray, d, wc_n, t);
```

Note that `t` is directly available through a depth map lookup (and a subsequent linearization).

Split Photon The splitting is done in a loop

```
// Rename for consistency with theory
vec3 wc_x = wc_position;
// BRDF
vec4 f = rho_d / PI; // [sr^-1]
// Splitting
for (int i = 0; i < N; i++)
    store_first_bounce_in_both_directions(
        i,
        photon,
        wc_x,
        wc_n,
        Phi_p * f);
```

We use the constant Lambertian BRDF. This could in principle be replaced with any BRDF. Also note that called procedure `store_first_bounce_in_both_directions` will actually store two photons (one in each trace direction). This is similar to the approach we choose for AO. The sub-steps of the this procedure are explained next.

(a) The trace function is completely analogous to the one used in Section 4.4.1. We do augment it to also store the positions (`wc_x_previous` and `wc_x_next`) of the intersected surface in each direction. This change is trivial.

(b) The projection is done using the `project_wc_to_sc` routine (Listing 4). We augment this routine to also return the photon's depth value in *eye coordinates* (EC) from the user's view (`ec_z_seen_by_user`). This depth is tested against the photon's actual depth value (`ec_z_actual`) to discard occluded photons. In the following, we use the previous intersection (`x_previous`) as an example:

```
// The actual depth value
float ec_z_actual = (user_view.view_matrix * vec4(
    wc_x_previous, 1.0)).z;
// x_previous in screen coordinates
ivec2 sc_x_previous;
// The observed depth value
float ec_z_seen_by_user;
// Only use visible photons
if (project_wc_to_sc(wc_x_previous, ldm_view,
    sc_x_previous, ec_z_seen_by_user)
    && ec_z_seen_by_user < ec_z_actual + const_bias)
{
    /* Use photon */
}
```

The `const_bias` variable is used to control threshold of the depth test. This is needed due to finite floating point precision.

(c, d) Diffuse reflection is done es explained in Section 5.3.1. While the rotation quaternion q is in principle four-dimensional, it can be interpreted as a tuple of a three-dimensional vector and the rotation angle. As such, it can be represented in GLSL as

```
vec4 q = vec4(v, theta);
```

where `v` is a `vec3` and `theta` is a `float`. Thus the α function can be implemented as follows

```
vec4 alpha( in vec3 w_i, in vec3 w_o ) {
    return normalize(vec4(
        cross(w_i, w_o),
        1.0 + dot(w_i, w_o)));
}
```

Similarly, quaternion rotation can be implemented just as presented in the formula

```
vec3 rotate_vector( vec4 q, vec3 v ) {
    return v + 2.0 * cross(
        q.xyz,
        cross(q.xyz, v) + q.w * v);
}
```

Note that no expensive trigonometric functions are used. Just cross products and dot products which maps well to graphics hardware.

The diffuse reflection of the photon differential is then straightforward. It is shown in Listing 9. Thus reflection and transferring just becomes calls to the corresponding procedures

```
// Rename for consistency with theory
vec3 w_i = normalize(-vertex.wc_view_ray_direction);
vec3 w_o = normalize(-ldm_view.forward);
vec3 wc_n_p = /* G-buffer lookup */;
float t = distance(wc_x, wc_x_previous);
// Reflect and transfer
diffusely_reflect(photon, w_i, w_o);
transfer(photon, w_o, wc_n_p, t);
```

Analogously, the same method can be applied to the photon traced in the other direction (corresponding to `wc_x_next`). In this case,

```
vec3 w_o = normalize(ldm_view.forward);
```

```
photon_differential construct_photon_differential( in vec3 d_hat, in vec3 right, in vec3 up ) {
    vec3 Du_d = (dot(d_hat, d_hat) * right - dot(d_hat, right) * d_hat) / pow(dot(d_hat, d_hat), 3.0 / 2.0);
    vec3 Dv_d = (dot(d_hat, d_hat) * up    - dot(d_hat, up   ) * d_hat) / pow(dot(d_hat, d_hat), 3.0 / 2.0);
    return photon_differential(vec3(0.0), vec3(0.0), Du_d, Dv_d);
}
```

Listing 7: *Construction of a photon differential.*

```
void transfer( inout photon_differential photon, in vec3 d, in vec3 n, in float t ) {
    float Du_t = -dot((photon.Du_x + t * photon.Du_d), n) / dot(d, n);
    float Dv_t = -dot((photon.Dv_x + t * photon.Dv_d), n) / dot(d, n);

    photon.Du_x = (photon.Du_x + t * photon.Du_d) + Du_t * d;
    photon.Dv_x = (photon.Dv_x + t * photon.Dv_d) + Dv_t * d;
}
```

Listing 8: *Transfer of a photon differential.*

Otherwise, the methods are the same.

(e) Recall that radiant flux is traced the same way as radiance. Thus we are still missing to compute the cosine term which accounts for projection. This is done next.

```
float cos_theta = dot(wc_normal, w_o); // [sr]
float Phi_p = Phi_p_and_f * cos_theta; // [W]
if (0.0 < cos_theta)
    store_photon(wc_next, wc_hit_normal, photon,
     Phi_p);
```

where `Phi_p_and_f` ($\Phi_p f_r$) was passed in earlier. Note that we only store the photon if it actually contributes. The photon differentials are stored in an SSBO called `photons`. The `store_photon` procedure is given in Listing 10.

Footprint Culling Note the `/* Footprint culling */` comment in Listing 10. This is an optional step which discards photon's whose footprint are larger than a certain threshold, T_{foot}. This is because large photon footprints have two negative properties: They are costly to splat and contribute little to the overall image. The footprint culling is done by testing the max norm of the differential against T_{foot}

```
// Footprint culling
// Find the max norm of the differential
vec3 abs_x = max(abs(photon.Du_x), abs(photon.Dv_x));
float max_x = max(max(abs_x.x, abs_x.y), abs_x.z);
// Discard large footprints
const float T_footprint = 0.5;
if (T_footprint < max_x) return;
```

This culling mechanism introduces additional bias in the algorithm. On the other hand, performance is improved. We investigate the impact of T_{foot} in Section 6.2.5.

Source Code The complete GLSL code can be found in the appendix.

5.4.2 Photon Splatting

The photon buffer generated in the previous pass is sent through the rasterization pipeline in the next pass. This is simply a matter of rebinding the underlying buffer object as a vertex array (instead of an SSBO) and issuing a draw call (as points). The properties stored through the previous SSBO binding can then be accessed as vertex attributes directly in the vertex shader. E.g.,

```
layout(location = 0) in vec3 wc_x;
layout(location = 1) in vec3 wc_n;
layout(location = 2) in vec3 Du_x;
layout(location = 3) in vec3 Dv_x;
layout(location = 4) in vec4 Phi; // [W]
```

This pass is rendered over a full-screen quad. The scene information is available through the G-buffer. The splatting itself is best explained in terms of the three shader stages:

1. **Vertex Shader.** The irradiance estimate is made here.
2. **Geometry Shader.** The positional differential is used to expand the point into a quad.
3. **Fragment Shader.** The kernel function, K, is applied and the result is written to the framebuffer. Additive blending is used so that the sum in Equation 27 is computed.

Please refer to Figure 28 for an overview. We will go into further details in the following paragraphs.

Vertex Shader First, the global scale parameter, s, is applied to the photon differential

```
photon.Du_x = Du_x * s;
photon.Dv_x = Dv_x * s;
```

In this context, `photon` refers to the vertex attributes which are sent to the next shader stage. At this point, we also set the previously-mentioned a parameter (controlling topological bias). In practice, we found that `a = scale` worked nicely. This also reduces the number of empirical parameters. Next, the M_p matrix (Equation 26) is computed

```
photon.M = mat3_from_rows(
    cross(photon.Dv_x, wc_n),
    cross(wc_normal, photon.Du_x),
    a * wc_n);
photon.M *= 2.0 / dot(
    photon.Du_x,
    cross(photon.Dv_x, wc_n));
```

Note that we use a custom matrix constructor, `mat3_from_rows`, since the default `mat3` constructor is column-wise. We also use include the third row of M which has the topological bias parameter

```
void diffusely_reflect( inout photon_differential photon, in vec3 w_i, in vec3 w_o ) {
    vec4 q = alpha(w_i, w_o);
    photon.Du_d = rotate_vector(q, photon.Du_d);
    photon.Dv_d = rotate_vector(q, photon.Dv_d);
}
```

Listing 9: *Diffusely reflect of a photon differential.*

```
void store_photon( in vec3 wc_x, in vec3 wc_n, in photon_differential photon, in vec4 Phi_p ) {
    /* Footprint culling */

    uint32_t id = atomicCounterIncrement(photon_count);

    photons[id].wc_x = vec4(wc_x, 1.0);
    photons[id].wc_n = vec4(wc_n, 0.0);
    photons[id].Du_x = vec4(photon.Du_x, 0.0);
    photons[id].Dv_x = vec4(photon.Dv_x, 0.0);
    photons[id].Phi  = Phi;
}
```

Listing 10: *Storing photon differentials.*

(a). As mentioned earlier, this row can be omitted for added performance. Alternatively, the angle between the photon's recorded surface normal (`wc_n`) can be compared directly to the actual surface normal in a later step. E.g., using a dot between between the normals. We choose to include topological bias directly in M since graphics hardware is optimized for matrix vector multiplication (and not for comparisons).

Lastly, the irradiance estimate (Equation 24) is made

```
float A_p = PI / 4.0 * length(cross(
    photon.Du_x,
    photon.Dv_x)); // [m^2]
photon.E = Phi / A_p; // [W * m^-2]
```

Geometry Shader The geometry shader takes a single point as input and emits a quad (in the form of a triangle strip). The input vertex has the photon data

```
in photon_data {
    vec3 wc_x, Du_x, Dv_x;
    vec4 E;
    mat3 M;
} photon[];
```

Note that this is declared as an array even though there is only a single point. This is merely a convention imposed by GLSL. The geometry shader emits vertices that we refers to as splats

```
out splat_data {
    flat vec3 wc_position;
    flat mat3 M;
    flat vec4 irradiance;
} splat;
```

All attributes are declared as `flat` since no interpolation is needed. The quad generation itself is straight-forward but lengthy. It is given in Listing 11. We use the helper function `cc_position` to project the vertices into *clip coordinates* (CC).

Fragment Shader Lastly, the splat is actually written to the framebuffer

```
// Get scene properties
vec3 wc_position = /* G-buffer lookup */
vec4 rho_d = /* G-buffer lookup */
// Radiance estimate
float l = length(splat.M *
    (wc_position - splat.wc_x));
vec4 f = rho_d / PI;
L_o = PI * K(l) * f * splat.E;
```

Note that we use a Lambertian BRDF. As stated earlier, this can easily be replaced with BRDF. The resulting radiance is written directly to the framebuffer

```
layout(location = 0) out vec4 L_o;
```

Source Code Please refer to the appendix for the full source code.

5.4.3 Skipping the Photon Buffer

In Section 5.3.3, we hinted that the photon buffer could be skipped entirely. This was actually our first approach. Unfortunately, it turns out that splatting to an SSBO is very inefficient on current graphics hardware. Thus this approach was not viable in practice. We hope that future improvements in GPU architecture will improve the situation. As we will soon see, photon splatting is still the bottleneck of our indirect lighting method.

```
vec4 cc_position( in vec3 wc_offset )
{ return view_projection_matrix * vec4(photon[0].wc_position + wc_offset, 1.0); }

void main() {
    splat.wc_position = photon[0].wc_position;
    splat.irradiance = photon[0].irradiance;
    splat.M = photon[0].M;

    gl_Position = cc_position(0.5 * (-photon[0].Du_x - photon[0].Dv_x));
    EmitVertex();

    gl_Position = cc_position(0.5 * (-photon[0].Du_x + photon[0].Dv_x));
    EmitVertex();

    gl_Position = cc_position(0.5 * ( photon[0].Du_x - photon[0].Dv_x));
    EmitVertex();

    gl_Position = cc_position(0.5 * ( photon[0].Du_x + photon[0].Dv_x));
    EmitVertex();
}
```

Listing 11: *Converting photon differentials into splat quads.*

6 Results and Findings

In this section, we will evaluate our implementation. The evaluation will be both in terms of performance but also correctness. The former is measured quantitatively in milliseconds whereas the later is done qualitatively against a path traced reference. For the AO, we will also compare our implementation against HBAO. First, we evaluate the AO method. Second, we evaluate the indirect lighting method. Third, we show a combination of the two approaches.

Evaluation Settings All images are rendered in 800×800 resolution. We encourage readers of the PDF file to zoom in and see the images in full resolution. We have performed all tests on two different Nvidia GPUs: The GeForce GTX 480 and GeForce GTX 780 Ti. Please refer to Table 2 for additional details.

Note that for the GTX 780 Ti, it was necessary to use a memory barrier in the layered depth map construction pass (see Section 3.4.2). This is to be expected as the memory barrier is theoretically required. Furthermore, for some configurations it was also necessary to use the previously-mentioned `atomicAdd` workaround due to driver issues (see Section 3.4.3). We have marked tests using the workaround with an asterisk (*). We describe the impact of the workaround in Section 6.3.

6.1 Ambient Occlusion

First, we scale the attenuation parameter (d_{max}) to see how well our method captures local as well as global scene information. Second, we scale the number of layered depth maps (N_{LDM}) to see how image quality can be traded for performance. Third, we scale the resolution of the layered depth maps (R_{LDM}) while using an unattenuated visibility term to see how close we can get to the path traced reference. Fourth, we test the normal offset used to remove artifacts for thin surfaces.

The HBAO method has the following parameters: Number of directions traced (N_{HBAO}) and the number of ray-marching steps in each direction (S_{HBAO}). Furthermore, we use the variable M to denote the memory usage of the `data` buffer in which the list nodes and head pointers are stored.

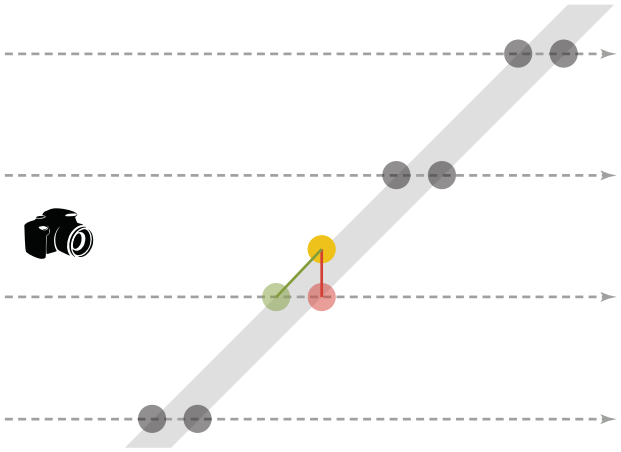


Figure 29: *Sampling error due to thin geometry. A sample position (yellow circle) is being traced in a layered depth map. The projection has correctly found the nearest L_p sequence. Now, the L_p sequence is being traversed to find the nearest sample. However, the nearest sample (red circle) actually belongs to the backface. Thus the tracing algorithm fails to find the real intersection (green circle). This happens when the geometry is thin and the surface normal is at an oblique angle to the layered depth map (as pictured).*

For each test, we provide both the overall frame time (written in bold) and the sub-timings of individual passes (written underneath). We have not included sub-timings for all aspects of rendering but only for the major passes. E.g., the G-buffer pass has been omitted. Thus the sub-timings do not necessarily add up to the overall time.

6.1.1 Scaling d_{max}

We have used an exponential attenuation function

$$V(x, \omega_i, d) = \min\left(\frac{d}{d_{max}}, 1\right)^2$$

since this is also what is implemented in the path traced reference. The HBAO method uses deterministic sampling directions just like the layered depth maps (to get a better comparison). We have tested d_{max} using the values 80 cm, 160 cm, 320 cm, 640 cm, and 1280 cm (Table 3). All other parameters are kept constant (see the caption). We have chosen a relatively low N_{LDM} in order to get reasonable performance.

It is clearly evident that our approach is closer to the path traced reference for large d_{max}. This is to be expected since HBAO is limited to the information available in the depth map. As such, HBAO does not have the same global scene information available as can be found in the layered depth maps. However, HBAO is indeed the fastest of the too approaches even for large d_{max}. It's somewhat surprising how well HBAO scales in terms of performance since larger d_{max} implies that texture fetches are further apart (causing cache misses). Still, we only observe a 1 ms difference from the $d_{max} = 80$ cm to $d_{max} = 640$ cm. In fact, HBAO's performance improves for $d_{max} = 1280$ cm. We hypothesize that this is because the ray-marching distance is so large that depth map lookups are attempted outside the visible region (thus not resulting in an actual texture fetch).

For HBAO, performance is consistent across both cards. Of course, the GTX 780 Ti is notably faster (being a newer card with better

Name	Dedicated VRAM	L2 Cache	Memory Bandwidth	Driver	Platform
GeForce GTX 480	1536 MB GDDR 5	768 KB	177.41 GB/s	Driver 344.75	Windows 8.1
GeForce GTX 780 Ti	3072 MB GDDR5	1536 KB	336.0 GB/s	Driver 347.25	Windows 7

Table 2: *Hardware configuration.*

specifications). For our approach, we do see an interesting difference. The GTX 480 uses the majority of the frame time on layered depth map construction. On the GTX 780 Ti, the construction time is equal to the time spent actually tracing AO. Moreover, construction on the GTX 780 Ti is approximately a factor 3 times faster than on the GTX 480. It is difficult to attribute this difference to any specific hardware difference. We hypothesize that the improved memory bandwidth helps tremendously. Specifically, since storing the list nodes may cause many cache misses. Of course, the larger L2 cache also helps to reduce cache misses overall. Note that performance is also completely independent of d_{max}. This is opposed to traditional SSAO methods where a larger d_{max} means a larger sample radius and thus worse performance. Though as mentioned above, this is seemingly not really an issue for HBAO.

In direct comparison, there is no doubt that HBAO is faster than layered depth maps. More than twice as fast consistently. Moreover, HBAO produces fewer artifacts in that time. Banding artifacts are clearly visible in our methods even for $d_{max} = 80$ cm. We could use a smaller N_{LDM} to improve performance but the artifacts would become even worse. At the current level, we postulate that the artifacts will be somewhat hidden by texture details and direct lighting. If not, a blurring pass can be applied though this requires additional frame time.

Our method only shines on one front: correctness. If physically-based rendering is a key priority, then our method is the better choice. Performance-wise, we hope to see improvements on newer hardware of the same magnitude observed between the GTX 480 and the GTX 780 Ti. Note that the total memory use is only $M = 25.24$ MB. As such, memory requirements do not seem to be a problem for this configuration. Likewise, the clear pass is negligible in the overall frame time.

6.1.2 Scaling N

The main source of artifacts in our method are due to undersampling the integral. Thus we scale N to see the effect on image quality and performance. We have tested N using the values 32, 64, 128, 256, and 512 (Table 4). To make the comparison fair, we scale both N_{LDM} and N_{HBAO} simultaneously.

In our method, the image quality is clearly improved for larger N . For $N = 512$ it is very close to the reference. Still, low scale geometric details are not captured. This is especially visible in the plants which lack shadowing. This is due to the limited resolution, $R_{\text{LDM}} = 200 \times 200$. That is, the layered depth map are to coarse to distinguish between individual leaves. This is also due to the sponza being a large scene. I.e., each layered depth map must be spread out to cover the whole scene thus reducing the small-scale definition. Another noticeable artifact is the banding in the floor. This is due to the deterministic sampling approach. As such, only large N can improve on the situation.

HBAO still struggles to capture large scale details. The only real improvement is that there are fewer banding artifacts for large N. It is clearly evident that HBAO does not produce correct results even for large N. Note that we have even set $S_{\text{HBAO}} = 128$ which is very costly. Notice the wrong shadows between the pillars and the chains. These are caused by missing information. Specifically, that the depth map does not contain information about the other side of the chain. That is, HBAO does not know that the chain is actually a small occluder. In conclusion, the HBAO approximations are too rough to converge to the physical correct result for large d_{max}. Still, small-scale effects are captured nicely.

In both approaches, the improved image quality comes at a performance cost. Specifically, the frame time is approximately directly proportional to N on both GPUs. Again, we observe the same trend that the construction dominates on the GTX 480 whereas it is equal to the AO computation for the GTX 780 Ti. The same explanation applies. Note that not only is the overall frame time proportional to N; all sub-timings are also proportional. This makes sense since each step is implemented as a loop over N. The memory usage, M, also increases proportional to N for the same reasons. Specifically, $M = 805$ MB for $N = 512$. This is clearly an impossible memory requirement for real-time applications since there must also be room for textures, models, animations and so on. However, at a frame time of 512.4 ms this configuration can hardly be called real-time anyhow. As such, we suggest that configurations with large N should be reserved for either interactive or offline purposes.

6.1.3 Unattenuated Visibility and Scaling R_{LDM}

We test with an unattenuated visibility function, $V(x, , \omega_i)$, such as used in the physically correct version of AO. Since the ray-marching length in HBAO depends on d_{max}, we cannot compare against HBAO in this test. This is also another indication that HBAO is not physically correct.

Furthermore, we noted that scaling N produced good quality. Thus we now scale R_{LDM} to see the resolutions effect on performance and image quality. We test R_{LDM} using the values 50×50, 100×100, 150×150, and 200×200 (Table 5). Furthermore, we also test our approach on the hairball model which is typically used in AO comparisons.

The first observation is that R_{LDM} affects the memory usage. One would normally expect that doubling the resolution leads to four times the memory requirements, M. However, this is not observed. At most a doubling of M is observed and often lower. We hypothesize that this is due to uneven depth complexity. That is, the memory usage is not evenly distributed over the layered depth map (as it would be for a regular depth map). This hypothesis is backed by the fact that for the sponza (which has very uneven depth complexity), M consistently increases very little (less than double) each time R_{LDM} is doubled. In contrast, the hairball (which has a more even depth complexity) sees a large increase in M (sometimes more than double) each time R_{LDM} is doubled.

The memory usage alone is also interesting. Note that rendering the hairball was not even possible on the GTX 480 for $R_{\text{LDM}} = 150 \times 150$ and beyond. This is because the layered depth maps simply do not fit into the available memory. The GTX 780 Ti was able to render the hairball for large R_{LDM} since it has double the memory of the GTX 480. This is also the reason that we did not test resolutions beyond $R_{\text{LDM}} = 200 \times 200$: Current GPUs simply do not have enough memory. Of course, one can use a lower N but

		Layered Depth Maps		Reference	HBAO	
		Time	Image	Image	Image	Time
$d_{max} = 80\,\text{cm}$	**GTX 480**	**36.68 ms**				**15.38 ms**
	Clear	0.5445 ms				
	Construction	18.61 ms				
	AO	10.33 ms				8.116 ms
	GTX 780 Ti	**14.43 ms**				**5.319 ms**
	Clear	0.2518 ms				
	Construction	5.810 ms				
	AO	5.913 ms				2.837 ms
$d_{max} = 160\,\text{cm}$	**GTX 480**	**36.61 ms**				**15.87 ms**
	Clear	0.5715 ms				
	Construction	18.63 ms				
	AO	10.35 ms				8.438 ms
	GTX 780 Ti	**14.47 ms**				**5.770 ms**
	Clear	0.2620 ms				
	Construction	5.842 ms				
	AO	5.928 ms				3.180 ms
$d_{max} = 320\,\text{cm}$	**GTX 480**	**36.71**				**16.13 ms**
	Clear	0.5766 ms				
	Construction	18.62 ms				
	AO	10.32 ms				8.979 ms
	GTX 780 Ti	**14.57 ms**				**6.046 ms**
	Clear	0.2602 ms				
	Construction	5.872 ms				
	AO	5.929 ms				3.559 ms
$d_{max} = 640\,\text{cm}$	**GTX 480**	**36.72 ms**				**16.88 ms**
	Clear	0.5626 ms				
	Construction	18.59 ms				
	AO	10.35 ms				9.624 ms
	GTX 780 Ti	**14.66 ms**				**6.585 ms**
	Clear	0.2623 ms				
	Construction	5.978 ms				
	AO	5.938 ms				3.661 ms
$d_{max} = 1280\,\text{cm}$	**GTX 480**	**36.39 ms**				**16.77 ms**
	Clear	0.5534 ms				
	Construction	18.52 ms				
	AO	10.36 ms				9.554 ms
	GTX 780 Ti	**14.53 ms**				**5.764 ms**
	Clear	0.2587 ms				
	Construction	5.866 ms				
	AO	5.920 ms				3.245 ms

Table 3: *Scaling d_{max}. Constant parameters: $N_{LDM} = 16$, $R_{LDM} = 200 \times 200$, $N_{HBAO} = 20$, and $S_{HBAO} = 8$. Constant metrics: $M = 25.24$ MB.*

			Layered Depth Maps		Reference	HBAO	
			Time	Image	Image	Image	Time
$N = 32$	$M = 50.44$ MB	**GTX 480**	**70.27 ms**				**189.3 ms**
		Clear	1.094 ms				
		Construction	39.93 ms				
		AO	20.98 ms				180.3 ms
		GTX 780 Ti	**28.28 ms**				**59.40 ms**
		Clear	0.5228 ms				
		Construction	13.06 ms				
		AO	12.10 ms				56.28 ms
$N = 64$	$M = 100.3$ MB	**GTX 480**	**126.2 ms**				**369.6 ms**
		Clear	2.042 ms				
		Construction	74.47 ms				
		AO	42.35 ms				360.9 ms
		GTX 780 Ti	**54.93 ms**				**115.3 ms**
		Clear	1.088 ms				
		Construction	26.51 ms				
		AO	24.78 ms				112.5 ms
$N = 128$	$M = 201.0$ MB	**GTX 480**	**247.2 ms**				**733.8 ms**
		Clear	4.096 ms				
		Construction	149.6 ms				
		AO	86.3 ms				722.9 ms
		GTX 780 Ti	**108.7 ms**				**227.8 ms**
		Clear	2.319 ms				
		Construction	52.94 ms				
		AO	50.9 ms				225.0 ms
$N = 256$	$M = 402.4$ MB	**GTX 480**	**490.6 ms**				**1465 ms**
		Clear	8.179 ms				
		Construction	300.2 ms				
		AO	174.8 ms				1450 ms
		GTX 780 Ti	**218.9 ms**				**454.7 ms**
		Clear	4.883 ms				
		Construction	105.7 ms				
		AO	105.4 ms				451.9
$N = 512$	$M = 805.5$ MB	**GTX 480**	**1024 ms**				**2904 ms**
		Clear	16.42 ms				
		Construction	644.5 ms				
		AO	354.5 ms				2894 ms
		GTX 780 Ti	**512.4 ms**				**903.6 ms**
		Clear	10.3 ms				
		Construction	279.8 ms				
		AO	215.5 ms				900.2 ms

Table 4: *Scaling N. Constant parameters: $R_{LDM} = 200 \times 200$, $S_{HBAO} = 128$, and $d_{max} = 1280$ cm.*

that comes at a loss in image quality.

In terms of correctness, we do see that R_{LDM} influences the results. However, whereas large N reduces banding artifacts, large R_{LDM} seems to enhance the existing artifacts. This is especially noticeable in the bands observed on the floor. Thus R_{LDM} is best kept low though this seems counter-intuitive. As such, R_{LDM} can actually be interpreted as a coarse blurring parameter. The banding artifacts can only be improved through larger N. We had hoped that larger R_{LDM} would lead to more definition in the low-scale details (e.g., the plants). Unfortunately, this is not the case within the limits that we can test R_{LDM} in practice.

The hairball is seemingly unaffected by the choice of R_{LDM}. Note that the layered depth map can be wrapped tighter around the hairball (in contrast to the large sponza). This is why the small-scale details of the individual hairs are actually captured. The sampling density is already more than sufficient even for small R_{LDM}. The only thing that increases with R_{LDM} is the render time.

Lastly, we note that the `atomicAdd` workaround was needed for the GTX 780 Ti for the hairball scene. Therefore, the results are actually slightly worse than they should be.

6.1.4 Normal Offset

Lastly, we would like to point out the effect the normal offset has on the result (Figure 30). The default offset is 10 which is also the value used in all other renderings. Note the artifacts on thing surfaces such as the curtains (Refer to Figure 29 for an explanation). The normal offset mitigates these artifacts. However, if the offset is too large (e.g., 20) then small-scale occlusion details are lost. This is a real concern since our method already struggles with low-scale details. Thus it is critical to find a good default for the scene in question.

6.2 Indirect Lighting

First, we scale N for the same reasons as before. Second, we test different values of the global footprint scale, s. Third, we scale R_{LDM}. Fourth, we scale the resolution of the light rendering (R_{light}) and thus implicitly the number of primary photons. This is done simultaneously with s in an attempt to find an optimal combination. Fifth, we show the effect of the various threshold parameters. Sixth, we decouple the topological bias parameter (a) from s and show a's effect in isolation.

All scaling tests are done in the sponza using two different light setups. The sub-timings are: Buffer clearing, glsldm construction, photon tracing, and photon splatting.

6.2.1 Scaling N

We test N using the values 8, 32, 128, and 512 (Table 6). Like in AO, low values of N leads to banding artifacts. As N in increased, the image convincingly resembles the reference. Though some differences are noticeable. Notably in the first floor scene where our renders seem darker than the reference. This is because we only trace the first bounce of indirect light. The path traced reference traces multiple bounces (terminating light paths via Russian roulette). Therefore, it is expected that our solution is slightly darker. Still the resemblance is visually convincing which confirms that the first couple of light bounces are much more important than later bounces.

Due to the low frequency of indirect light, banding artifacts are not as big an issue as they where in AO. The worst artifacts are due to the photon splats not being large enough and thus not properly covering the scene. Another problem is light leaking around corners. This is especially noticeable on the box-shaped column in the first floor render. This is due to the photon splat covering both sides of the column. The topological bias reduces the effect but it is never completely gone. In contrast, the path traced reference always has clearly defined shadow edges. Similarly, light can also leak into occluded areas. E.g., under the carpets in the first floor render. In our approach, said carpets project a very soft shadow due to indirect light on the wall. In the reference, the same shadow has a much harder edge. The same artifact are even worse for the corresponding carpets in the sponza ground floor render.

Topological bias can mitigate light creeping around corners. However, on flat surface, such as the wall behind the carpets, there is no quick remedy. The only solution is to use a smaller scale, s, and compensating by emitting more photons. Of course, this comes at a performance cost.

Speaking of performance, our indirect lighting approach is only barely real-time for $N = 8$. At this level, however, the artifacts are too severe and the resulting image is not convincing. For $N = 32$, the artifacts are tolerable but the frame time has been increased by a factor of four. That is, the frame time is directly proportional to N even for the sub-steps (just as with AO). The reasoning is the same: All sub-steps are loops over N. We suggest that our indirect lighting method is used for interactive and offline purposes due to these performance characteristics.

Interestingly, neither construction or photon tracing are the dominant factors in the frame time. Instead, it is the photon splatting step which by far outweighs the other sub-timings. This is somewhat to be expected. As we mentioned previously, we attempted to implement splatting directly a fragment shader. We found that this approach was inefficient in practice. Therefore, we switched to use the fixed-function pipeline's additive blending mode. Still, the results show that splatting is the bottleneck. We hypothesize that the problem is that many splats map to the same image location. That is, multiple fragments compete to write to the same memory locations. As such, the pipeline is forced to serialize the writes which significantly reduces performance.

The GTX 780 Ti is twice as fast as the GTX 480 on average. This is to be expected. We already mentioned that the construction step is sometimes three times as fast on the GTX 780 Ti. Unfortunately, said step is not the main bottleneck. The splatting step is barely twice as fast which compensates for the fast construction step in the overall frame time. This also suggests GPU development has been focused on optimizing shader execution (which helps construction) but not the fixed function pipeline features such as additive blending (as used in splatting). Thus we do not expect to see significant improvements on future hardware. Instead, we hope that RMW in shader will be further improved so that single-pass fragment shader splatting becomes feasible.

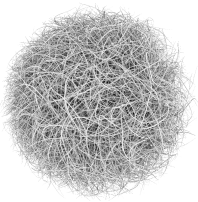
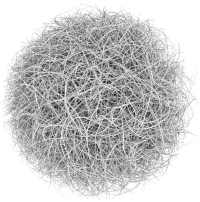
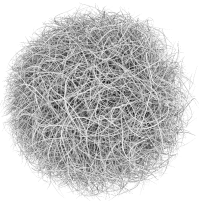
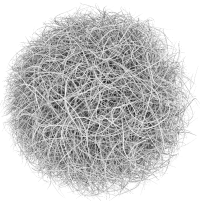
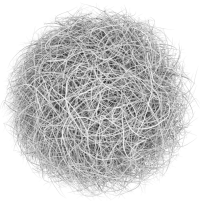
Lastly, we note that the memory requirements for the layered depth maps, M_{lists}, clearly dominate. The photon buffer size, M_{photons}, is relatively small (under 100 MB) even for large N.

6.2.2 Scaling s

Recall that s controls the size of the photon differentials. Also, the topological bias (a) is bound to s. We test s using the values 200, 400, 800, and 1600 (Table 7). It is clear, that s has a big impact on both image quality and performance. For small s, the photons can be made out individually as splotches of light. E.g., the faded red splotches on the ground floor. For large s, the solution convincingly resembles the reference image. Though, as noted before, light may creep into places where it shouldn't because of the large splat size. Ideally, N should be much larger while s should be small. E.g.,

$R_{\text{LDM}} = 50 \times 50$	$R_{\text{LDM}} = 100 \times 100$	$R_{\text{LDM}} = 150 \times 150$	$R_{\text{LDM}} = 200 \times 200$	Reference
				
793.2 ms	**810.9 ms**	**875.4 ms**	**983.9 ms**	**GTX 480**
5.728 ms	6.621 ms	10.71 ms	16.41 ms	Clear
446.4 ms	459.3 ms	507.3 ms	604.3 ms	Construction
332 ms	335.7 ms	349.6 ms	354.4 ms	AO
373.0 ms	**386.0 ms**	**440.1 ms**	**454.8 ms**	**GTX 780 Ti**
2.324 ms	3.613 ms	6.490 ms	10.14 ms	Clear
176.2 ms	180.7 ms	217.6 ms	228.4 ms	Construction
191.5 ms	198.0 ms	209.3 ms	213.5 ms	AO
204.1 MB	324.2 MB	525.1 MB	805.5 MB	M_{Lists}

(a) *Sponza ground floor.*

$R_{\text{LDM}} = 50 \times 50$	$R_{\text{LDM}} = 100 \times 100$	$R_{\text{LDM}} = 150 \times 150$	$R_{\text{LDM}} = 200 \times 200$	Reference
				
3561 ms	**3826 ms**	**N/A**	**N/A**	**GTX 480**
7.007 ms	14.3 ms			Clear
2727 ms	2774 ms			Construction
806.7 ms	1017 ms			AO
1174 ms	**1812 ms**	**2768 ms**	**4439 ms**	**GTX 780 Ti***
3.139 ms	8.651 ms	17.66 ms	30.14 ms	Clear
738.7 ms	1286 ms	2111 ms	3543 ms	Construction
425.7 ms	511.4 ms	632.9 ms	859.3 ms	AO
295.5 MB	690.2 MB	1348 MB	2269 MB	M_{Lists}

(b) *Hairball.*

Table 5: *Scaling R_{LDM}. Constant parameters: $N = 512$.*

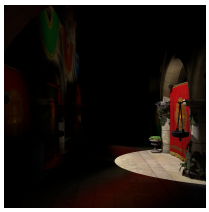
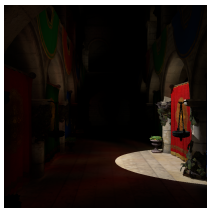
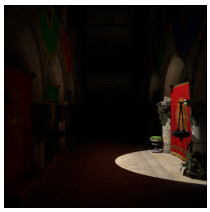
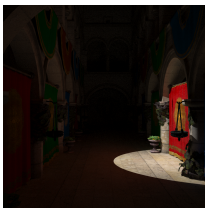
(a) *Offset scale 0 (no offset).*

(b) *Offset scale 1.*

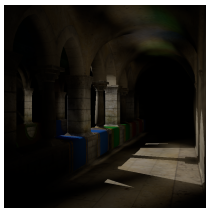
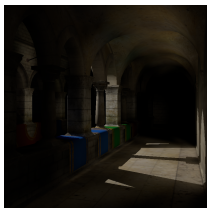
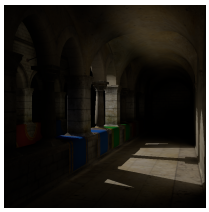
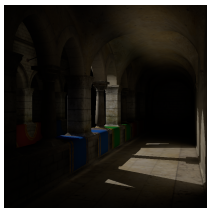
(c) *Offset scale 10 (default).*

(d) *Offset scale 20.*

Figure 30: *Scaling the normal offset. Constant parameters: $N = 512$, $R_{LDM} = 200 \times 200$, and $d_{max} = 200$ cm.*

$N = 8$	$N = 32$	$N = 128$	$N = 512$	Reference
				
81.88 ms	**328.3 ms**	**1237 ms**	**4919 ms**	**GTX 480**
0.338 ms	1.068 ms	4.229 ms	16.53 ms	Clear
9.395 ms	39.14 ms	156.8 ms	643.1 ms	Construction
0.5093 ms	1.338 ms	5.21 ms	19.84 ms	Tracing
62.81 ms	277.2 ms	1059 ms	4224 ms	Splatting
47.60 ms	**166.1 ms**	**618.1 ms**	**2394 ms**	**GTX 780 Ti**
0.2427 ms	0.5704 ms	2.431 ms	10.25 ms	Clear
3.879 ms	15.34 ms	72.17 ms	250.7 ms	Construction
1.100 ms	4.037 ms	15.19 ms	59.79 ms	Tracing
32.76 ms	136.5 ms	516.9 ms	2062 ms	Splatting
12.84 MB	50.44 MB	201.0 MB	805.5 MB	M_{lists}
1.175 MB	5.344 MB	20.97 MB	84.49 MB	M_{photons}

(a) *Sponza ground floor.*

				
68.22 ms	**220.2 ms**	**848.5 ms**	**3317 ms**	**GTX 480**
0.3561 ms	1.120 ms	4.125 ms	16.43 ms	Clear
10.31 ms	41.81 ms	158.6 ms	649.4 ms	Construction
0.6216 ms	1.59 ms	5.661 ms	22.47 ms	Tracing
45.23 ms	163.4 ms	668.2 ms	2616 ms	Splatting
33.37 ms	**113.1 ms**	**432.8 ms**	**1732 ms**	**GTX 780 Ti**
0.1708 ms	0.5599 ms	2.308 ms	10.25 ms	Clear
3.648 ms	14.60 ms	58.95 ms	243.8 ms	Construction
0.7508 ms	2.435 ms	8.974 ms	38.9 ms	Tracing
25.48 ms	92.22 ms	359.1 ms	1434 ms	Splatting
12.84 MB	50.44 MB	201.0 MB	805.5 MB	M_{lists}
0.6411 MB	2.653 MB	10.31 MB	42.29 MB	M_{photons}

(b) *Sponza first floor.*

Table 6: *Scaling N. Constant parameters: $R_{LDM} = 200 \times 200$ and $s = a = 2000$.*

many photons with a small footprint. However, this configuration is not performant.

s is also shown to have a direct impact on performance. Specifically, the splatting time more than triples every time s is doubled. This makes good sense, since s is used to scale the positional differential. As such, the splat quad's area (A_{quad}) should quadruple when s is doubled

$$A_{\text{quad}} = \|sD_u x \times sD_v x\| = s^2 \|D_u x \times D_v x\|$$

Fortunately, we do not quite observe quadrupled splatting time. This can be due to caching effects. The trend, however, is towards quadratic as s is increased. We hypothesize that the frame splatting will scale quadratically for larger s (when the cache is effectively defeated). When s is low, splatting is fast and construction becomes the bottleneck instead. Thus scaling down N accordingly is seemingly a good option. As stated earlier, however, photon splotches are visually disturbing for small s. Thus scaling N down will only worsen the image quality.

In this case, the timings on both GPUs lead to the same conclusions. We did not note any significant differences. This further supports that the fixed-function pipeline including additive blending (as used in splatting) has not been the focus of recent development. Note that s only affects the splatting time and not the other sub-timings as expected. Similarly, the memory use is constant throughout this test.

6.2.3 Scaling R_{LDM}

The resolution of each layered depth map (R_{LDM}) is scaled to see the response in image quality and performance. We test R_{LDM} using the values 50×50, 100×100, 150×150, and 200×200. As with AO, the result are counter-intuitive. Seemingly, increasing R_{LDM} has a positive effect on performance. This can be explained by observing the photon buffers memory use. The larger R_{LDM}, the smaller M_{photons}. That is, fewer photons are stored for large layered depth map resolutions. Splatting is the main bottleneck so smaller M_{photons} implies fewer photons to splat and in turn improved performance. These results are consistent across both test setups and GPUs. We hypothesize that larger R_{LDM} implies more fine-grained photon culling which in turn results in fewer photons overall. The absolute difference in M_{photons} is actually small. Since splatting is slow, however, even a small difference in the number of photons can bee directly observed in the overall frame time. In fact, the construction time actually increases with R_{LDM} but the reduced splatting time greatly compensates for this. Again, this confirms that splatting is the main bottleneck.

For both GPUs, the optimal value for R_{LDM} seems to be around 150×150. This is clearly the case in the first floor scene. For the ground floor, values beyond 150×150 has less of an impact on performance (in contrast to smaller values). This finding may vary from scene to scene.

In terms of image quality (and correctness), R_{LDM} has no significant impact. This is in line with our earlier findings modulo the banding artifacts (which are not a real problem for indirect lighting).

6.2.4 Scaling R_{light} and $s = a$

We scale the resolution of the light source used in photon tracing (R_{light}) which directly influences the number of primary photons (and thus the number of photons overall). The default setting used in the above tests is $R_{\text{light}} = 100 \times 100$. Recall that the number of primary photons is actually $n_{e,p} = \frac{\pi}{4} R_{\text{light}}$ due to spot light culling. Simultaneously, we scale s (and therefore a) to compensate for the increase or decrease in the photon count. The purpose is to empirically find an optimal combination of the two variables. In the above tests, we generally used $s = a = 2000$ which was conservative. Now we find the minimum value. The results are in Table 9.

It is immediately clear that a smaller s can be used while retaining much of the image quality. In fact, even when R_{light} is smaller than the default value (100×100), s can be set to 800 without much loss in image quality. Still, the result is visibly more splotchy though this is hardly noticed due to texturing. More importantly, performance is significantly improved for all configurations overall. Now, the splatting time is no longer dominant for the majority of the tests.

As R_{light} is increased, we see an increase in M_{photons} too as expected. While M_{photons} is still a modest quantity, this small increase combined with the constantly big M_{lists} was enough to defeat the GTX 480 in the most taxing test. Again, this is due to memory limits. Note that doubling R_{light} does not imply that s can be halved. On other words, a large number of additional photons are needed to compensate for even a small decrease in s. We have attempted to keep the image quality constant when tweaking s. The relationship between the number of photons (and hence R_{light}) and s is involved and not easily derived for complex scenes such as the sponza. Therefore, the ideal combination of R_{light} and s must be found empirically. Note that N also influences the number of photons. We have kept N fixed to both simplify the test and in order to ensure a good hemispherical approximation. That is, to avoid the latter influencing the result.

Besides performance, smaller s also mitigates some of the light creeping artifacts. Specifically, the indirect shadows (seen under the circular carpets) are much more refined. This is closer to the path traced reference. Still, the indirect shadows are a little too soft. Only in the limit $s \to 0$ will the indirect shadowing be perfect.

6.2.5 Footprint Threshold

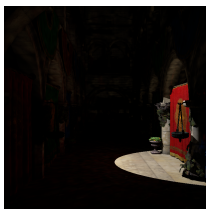
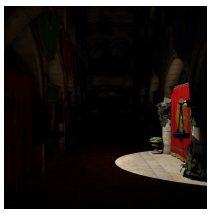
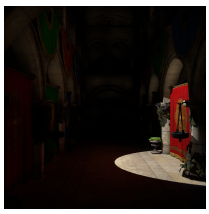
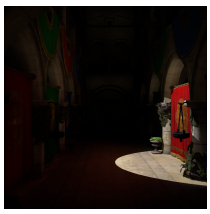
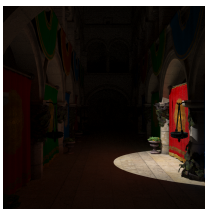
The footprint threshold (T_{foot}) is used to cull large photon footprints. The default value is 0.5. This introduces additional bias to the solution. The purpose of the footprint threshold is to reduce the time used to splat photons. Specifically, the time wasted splatting large photons with little contribution (due to the large footprint area). The results are in Figure 31.

On the far right of Figure 31 is the ground truth with no photon culling. On the far left, we use $T_{\text{foot}} = 0.1$ which results in more than half of the photons being culled (compare M_{photons} to the ground truth). Of course, splatting performance is significantly improved (almost by a factor of four) but the culling is too severe. Almost no indirect lighting is present in the resulting image. Using a conservative $T_{\text{foot}} = 5.0$ only culls 0.38 % of the photons. Correspondingly, performance is improved by 2.0 %. The default setting of $T_{\text{foot}} = 0.5$ is our chosen middle ground. Only 5.4 % of the photons are culled whereas the performance is improved by 20 %.

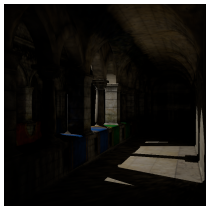
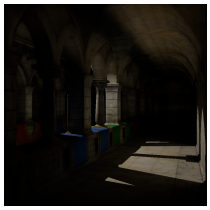
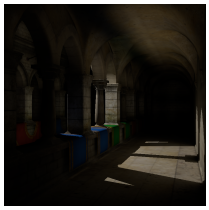
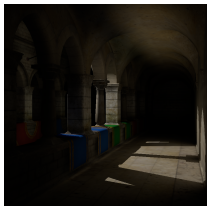
Since splatting is a large factor in the overall frame time, we feel that it is necessary to trade performance for bias via T_{foot}. For physical correctness, this step should be omitted.

6.2.6 Depth Culling Threshold

Recall that photons are also culled when projected into the users view. In this process a depth threshold is used because the depth

$s = a = 200$	$s = a = 400$	$s = a = 800$	$s = a = 1600$	Reference
				
710.9 ms	**901.3 ms**	**1475 ms**	**3337 ms**	**GTX 480**
16.42 ms	16.46 ms	16.55 ms	16.46 ms	Clear
602.4 ms	643.1 ms	686.6 ms	602.3 ms	Construction
19.8 ms	19.48 ms	19.49 ms	19.58 ms	Tracing
62.7 ms	210.9 ms	740.8 ms	2689 ms	Splatting
369.2 ms	**445.6 ms**	**716.4 ms**	**1622 ms**	**GTX 780 Ti**
10.38 ms	10.24 ms	10.25 ms	10.14 ms	Clear
255.0 ms	256.0 ms	253.0 ms	197.7 ms	Construction
59.99 ms	60.41 ms	60.93 ms	60.06 ms	Tracing
32.05 ms	108.3 ms	381.1 ms	1350 ms	Splatting

(a) *Sponza ground floor.*

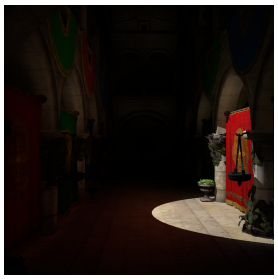
$s = a = 200$	$s = a = 400$	$s = a = 800$	$s = a = 1600$	Reference
				
690.4 ms	**774.8 ms**	**1097 ms**	**2335 ms**	**GTX 480**
16.43 ms	16.46 ms	16.47 ms	16.41 ms	Clear
605.7 ms	604.7 ms	603.8 ms	603.6 ms	Construction
22.41 ms	22.39 ms	22.41 ms	22.43 ms	Tracing
35.72 ms	121.0 ms	443.6 ms	1682 ms	Splatting
308.9 ms	**352.4 ms**	**549.8 ms**	**1221 ms**	**GTX 780 Ti**
10.22 ms	10.17 ms	10.38 ms	10.24 ms	Clear
240.9 ms	240.9 ms	250.0 ms	231.8 ms	Construction
36.63 ms	36.02 ms	35.80 ms	37.46 ms	Tracing
17.45 ms	61.66 ms	249.9 ms	937.6 ms	Splatting

(b) *Sponza first floor.*

Table 7: *Scaling s and a. $N = 512$ and $R_{LDM} = 200 \times 200$.*

(a) *Threshold* 0.1. *Splatting takes 1026 ms and $M_{photons} =$* 43.97 *MB.*

(b) *Threshold* 0.5. *Splatting takes 4186 ms and $M_{photons} =$* 84.49 *MB.*



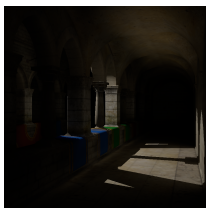
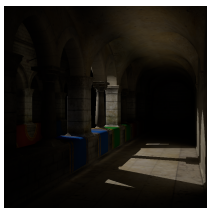
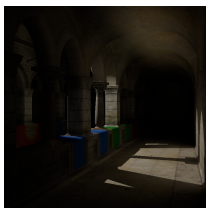
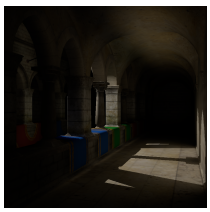
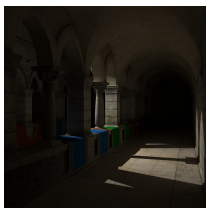
(c) *Threshold* 5.0. *Splatting takes 5111 ms and $M_{photons} =$* 88.99 *MB.*

(d) *No threshold. Splatting takes 5214 ms and $M_{photons} =$* 89.39 *MB.*

Figure 31: *Scaling the footprint threshold. Constant parameters: $N = 512$, $R_{LDM} = 200 \times 200$, and $s = a = 2000$.*

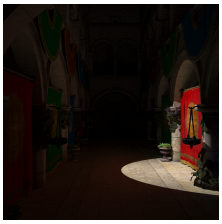
$R_{\text{LDM}} = 50 \times 50$	$R_{\text{LDM}} = 100 \times 100$	$R_{\text{LDM}} = 150 \times 150$	$R_{\text{LDM}} = 200 \times 200$	Reference
				
7058 ms	**6354 ms**	**5056 ms**	**4892 ms**	**GTX 480**
5.657 ms	6.8 ms	10.70 ms	16.51 ms	Clear
417.6 ms	474.2 ms	508.5 ms	650.7 ms	Construction
16.27 ms	17.7 ms	19.36 ms	19.55 ms	Tracing
5607 ms	4836 ms	4507 ms	4194 ms	Splatting
3027 ms	**2606 ms**	**2458 ms**	**2413 ms**	**GTX 780 Ti**
2.099 ms	3.607 ms	6.362 ms	10.35 ms	Clear
153.2 ms	149.9 ms	176.6 ms	255.3 ms	Construction
51.38 ms	55.25 ms	60.97 ms	59.34 ms	Tracing
2816 ms	2392 ms	2210 ms	2078 ms	Splatting
204.1 MB	324.2 MB	525.1 MB	805.5 MB	M_{lists}
99.15 MB	92.11 MB	90.37 MB	84.49 MB	M_{photons}

(a) *Sponza ground floor.*

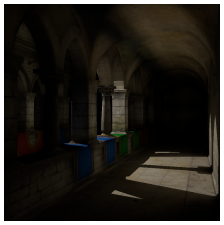
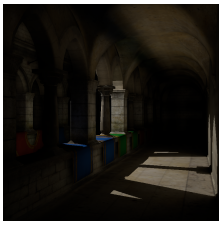
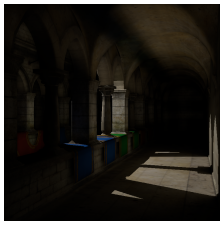
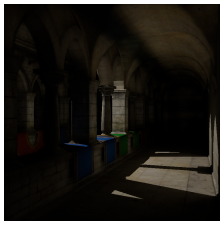
$R_{\text{LDM}} = 50 \times 50$	$R_{\text{LDM}} = 100 \times 100$	$R_{\text{LDM}} = 150 \times 150$	$R_{\text{LDM}} = 200 \times 200$	Reference
				
3725 ms	**3355 ms**	**3264 ms**	**3304 ms**	**GTX 480**
5.678 ms	6.727 ms	10.7 ms	16.51 ms	Clear
422.2 ms	445.4 ms	508.7 ms	649.7 ms	Construction
10.25 ms	19.80 ms	21.49 ms	22.56 ms	Tracing
3267 ms	2872 ms	2712 ms	2603 ms	Splatting
1900 ms	**1738 ms**	**1692 ms**	**1716 ms**	**GTX 780 Ti**
2.197 ms	3.679 ms	6.414 ms	10.17 ms	Clear
235.7 ms	239.0 ms	211.2 ms	240.1 ms	Construction
31.64 ms	34.39 ms	36.29 ms	36.00 ms	Tracing
1620 ms	1457 ms	1434 ms	1425 ms	Splatting
204.1 MB	324.2 MB	525.1 MB	805.5 MB	M_{lists}
46.35 MB	43.62 MB	42.84 MB	42.29 MB	M_{photons}

(b) *Sponza first floor.*

Table 8: *Scaling R_{LDM}. Constant parameters: $N = 512$ and $s = a = 2000$.*

$R_{\text{light}} = 50 \times 50$ $s = a = 800$	$R_{\text{light}} = 100 \times 100$ $s = a = 600$	$R_{\text{light}} = 150 \times 150$ $s = a = 500$	$R_{\text{light}} = 200 \times 200$ $s = a = 400$	Reference
				
869.2 ms	**1136 ms**	**1419 ms**	**N/A**	**GTX 480**
16.46 ms	16.55 ms	16.44 ms		Clear
642.6 ms	649.4 ms	644.2 ms		Construction
12.45 ms	19.73 ms	34.29 ms		Tracing
186.3 ms	438.8 ms	711.4 ms		Splatting
350.0 ms	**556.2 ms**	**766.4 ms**	**861.7 ms**	**GTX 780 Ti**
10.24 ms	10.28 ms	10.29 ms	10.15 ms	Clear
217.9 ms	249.5 ms	255.4 ms	215.5 ms	Construction
20.17 ms	60.31 ms	119.5 ms	191.0 ms	Tracing
97.66 ms	225 ms	370.4 ms	441.2 ms	Splatting
805.5 MB	805.5 MB	805.5 MB	805.5 MB	M_{Lists}
21.31 MB	84.49 MB	190.4 MB	338.3 MB	M_{Photons}

(a) *Sponza ground floor.*

$R_{\text{light}} = 50 \times 50$ $s = a = 800$	$R_{\text{light}} = 100 \times 100$ $s = a = 600$	$R_{\text{light}} = 150 \times 150$ $s = a = 500$	$R_{\text{light}} = 200 \times 200$ $s = a = 400$	Reference
				
828.4 ms	**920.8 ms**	**1126 ms**	**1178 ms**	**GTX 480**
16.46 ms	16.38 ms	16.47 ms	16.45 ms	Clear
666.4 ms	613.7 ms	651.3 ms	616.3 ms	Construction
13.78 ms	22.38 ms	34.84 ms	53.39 ms	Tracing
114.2 ms	257.9 ms	411.2 ms	482.0 ms	Splatting
329.8 ms	**507.0 ms**	**543.7 ms**	**614.1 ms**	**GTX 780 Ti**
10.18 ms	10.30 ms	10.24 ms	10.16 ms	Clear
240.2 ms	311.5 ms	239.9 ms	246.8 ms	Construction
14.30 ms	36.05 ms	73.17 ms	107.1 ms	Tracing
61.24 ms	138.1 ms	216.7 ms	245.6 ms	Splatting
204.1 MB	324.2 MB	525.1 MB	805.5 MB	M_{Lists}
10.73 MB	42.59 MB	95.73 MB	169.7 MB	M_{Photons}

(b) *Sponza first floor.*

Table 9: *Scaling R_{light} and $s = a$. Constant parameters: $N = 512$ and $R_{LDM} = 200 \times 200$.*

comparison has finite precision. The default is 0.1. The results are in Figure 32. The visual impact of the depth culling threshold is not directly noticeable. It can, however, be measured in the number of stored photons, M_{photons}. We conclude that finite precision is not actually a real concern in this case.

6.2.7 Scaling a

The topological bias is controlled via a. Previously, a was directly coupled to s. Now, we break this coupling and show the individual effect of a (Figure 33). The most apparent property is that large a is, the darker the scene becomes. This is because large a implies that light cannot creep around corners which results in less light overall. Of course, the intended effect of a is to just reduce this light creeping (without dimming the result). The problem is that the light, which is culled, is not compensated for. In theory, a should be removed altogether for physical correctness. However, a is needed in practice to ensure that the indirect light is not too smeared out.

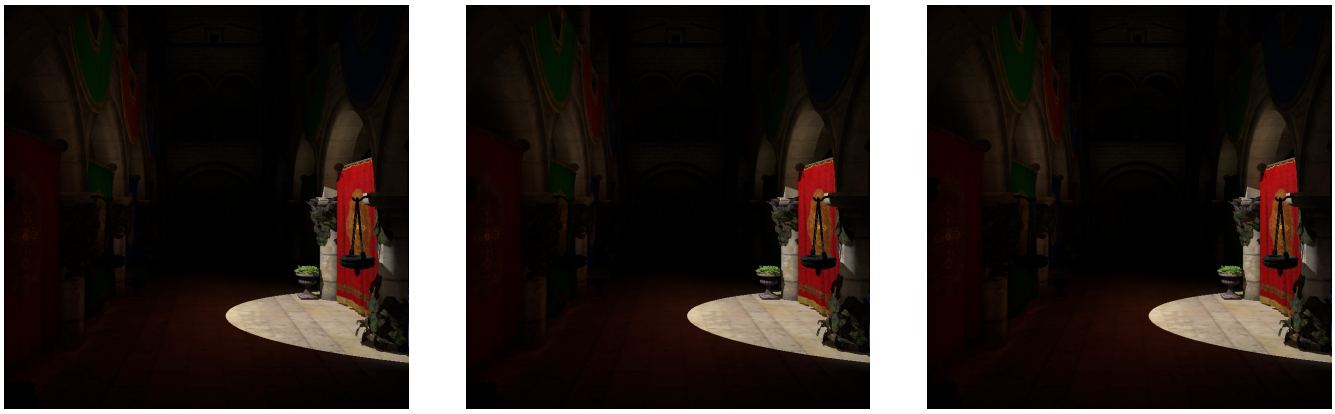
6.3 Impact of Workarounds

We test the impact of the `atomicAdd` workaround. The results are in Table 10. As noted earlier, an asterisk (*) denotes that the workaround has been used. From looking at the table, it is clear that the workaround has a significant impact. Performance is reduced by around 15–20 %. Fortunately, only one of our tests was affected by this. We hope that future drivers will fix this issue. For now, we just note that some configurations may not perform optimally due to this issue.

6.4 Combination

Lastly, we show that AO can be combined with indirect lighting. The idea is to weigh the environment light by the AO factor (Section 4.3.3) and combine this with the direct and indirect light of the spot light (Figure 34). The environment light gives a blue tint to the scene due to the illumination from the sky. Note that the environment light is only present in open areas whereas the indirect spot light traces into the corridors.

In this combination, the time spent on constructing layered depth maps is amortized by the tracing of environment light and indirect light. That is, the layered depth maps are constructed once per frame but can be used by both methods. This also shows that our auxiliary data structure based on layered depth maps is a multipurpose scene representation. See Section 7.4 for other uses of layered depth maps.

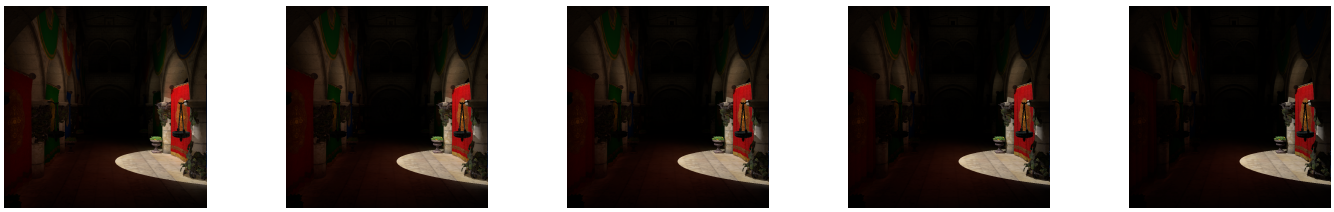


(a) *Threshold* 0.0. $M_{photons} = 83.95$ *MB*.

(b) *Threshold* 0.1 *(default)*. $M_{photons} = 84.49$ *MB*.

(c) *Threshold* 1.0. $M_{photons} = 89.08$ *MB*

Figure 32: *Scaling the depth culling threshold. Constant parameters:* $N = 512$, $R_{LDM} = 200 \times 200$, *and* $s = a = 2000$.



(a) $a = 200$.

(b) $a = 400$.

(c) $a = 800$.

(d) $a = 1600$.

(e) $a = 3200$.

Figure 33: *Scaling* a. *Constant parameters:* $N = 512$, $R_{LDM} = 200 \times 200$, *and* $s = 2000$.

	$N = 8$	$N = 32$	$N = 128$	$N = 512$
GTX 480	9.373 ms	37.07 ms	148.1 ms	603.5 ms
GTX 480*	11.48 ms	45.22 ms	181.1 ms	736.5 ms
GTX 780 Ti	3.061 ms	14.98 ms	61.81 ms	248.9 ms
GTX 780 Ti*	3.785 ms	17.50 ms	71.13 ms	288.6 ms

Table 10: *Impact of the* `atomicAdd` *workaround on layered depth map construction. Scaling* N *for the sponza ground floor.*

(a) *Sponza ground floor. Rendered in 6879 ms.*

(b) *Sponza first floor. Rendered in 3764 ms.*

Figure 34: *Direct environment light, direct spot light, and indirect spot light. Max settings.*

7 Discussion

First, our indirect lighting method is compared to other real-time approaches. Second, we list possible extensions and improvements to our method. Third, we suggest technical improvements for the implementation. Fourth, we list other uses for layered depth maps.

7.1 Indirect Lighting Comparison

In this section, we compare our indirect lighting method to previous work. First, we compare our method to VPL-based approaches. Second, we look at other layered depth map implementations.

7.1.1 Virtual Point Lights

Visibility One of the difficulties with real-time VPL methods is to evaluate the visibility term efficiently. Imperfect shadow maps is one solution but it is only approximate. Our indirect lighting method does not need to compute a visibility term since shadowing is implicitly handled via the photon tracing. As noted in the results, however, the indirect shadows in our approach might be biased due to light creeping around corners.

Bounces VPLs in general (not real-time generated) can easily produce multi-bounce light paths. The difficult part is to combine VPLs with a rasterization-based pipeline to get the benefits of both.

Real-time VPL methods such as reflective shadow maps are limited to one bounce of indirect light [Dachsbacher and Stamminger 2005]. Our method, as presented here, also has this limitation (see Section 7.2.1 for a trivial multi-bounce extension). So-called imperfect reflective shadow maps lifts this limitation and can theoretically be used for an arbitrary number of light bounces. Performance is reduced accordingly. As such, it would seem that current real-time VPL methods are limited to a single bounce of indirect light in practice. There is one exception: The VPL-path tracing hybrid mentioned earlier [Tokuyoshi and Ogaki 2012b]. This method efficiently produces $LDDDE$ paths (and can be trivially extended to more bounces).

Data Structure There are some key similarities between the reflective shadow map approach and our method. The reflective shadow maps are all stored in a single large texture. This is similar to how we store all link nodes in a single SSBO. Furthermore, each reflective shadow map is of low resolution. The authors suggests to use 128×128–256×256 maps [Ritschel et al. 2008]. Our layered depth map approach also uses a resolution in this range. Lastly, the basic idea of rendering the scene from multiple view is also implied in imperfect shadow maps.

BRDF In principle, any bidirectional reflectance distribution function can be used with our method (Section 7.2.2). Though the way we sample the hemisphere resembles Lambertian reflection the most. VPL-based methods assumes either diffuse or glossy surfaces. In principle, VPLs can also be used to represent specular reflections but they are not particularly suited for this purpose [Ritschel et al. 2008; Dachsbacher et al. 2014].

We have not tested other BRDFs besides the Lambertian. As such, we can only hypothesize about our methods ability to handle other reflection models. Still, it will be difficult to handle perfect specular reflections in our method since the outgoing directions are predetermined. Contrast this to conventional photon mapping which is very apt at handling caustics.

Light Source VPLs in general can be generated from arbitrary light sources. In real-time, however, reflective shadow maps must be generated be rendering the scene light's point of view. This is exactly the same approach that we employ in the **Photon Tracing** step. Thus both approaches are limited by the projection model (perspective, orthographic, etc.). In other words, both methods are limited to point-based or directional light sources (see Section 7.2.3 for a light source extension to our method). Such light sources are not physically-based since they do not have an area. This is a key limitation if physical correctness is a priority (for both methods).

Temporal Coherence In general, VPLs are generated using randomly generated light paths [Dachsbacher et al. 2014]. As such, these methods can have temporal flickering between frames due to changes in the sampling. A large number of VPLs is needed for temporal coherence which costs in terms of performance. The real-time VPL-based methods inherit these properties. Our approach uses deterministic sampling and as such has no problem with temporal flickering. The downside to our approach is the banding artifacts. As mentioned earlier, performance can be traded for fewer artifacts by sampling additional directions. The same principle applies to VPLs.

Quasi-random light paths can be used to generate more coherent light paths for the VPLs. This can be used to trade banding artifacts for noise. Moreover, to make the result temporally coherent. Note that reflective shadow maps can also be implemented completely deterministically [Dachsbacher and Stamminger 2005]. The problem is that deterministic screen-space sampling introduces banding artifacts.

Bias The main source of bias in VPL-based methods is due to bounding the G term. The stricter the bound, the dimmer the lighting. The exact same behaviour can be observed in our method with the a parameter (though for different reasons). The bound on G is intended to mitigate small lights clusters of great intensity. Such clusters are not generated in our method.

Our method has another source of bias: The irradiance estimate. The scale parameter, s, can used to control the extent of this bias. Unfortunately, s, must be rather large in practice to cover the scene adequately. The VPL-based methods do not have this problem.

In summary, both methods are biased. Said bias can be controlled via parameters for both VPLs and our approach. E.g., via the bound on G and s, respectively. Likewise, the image quality of both methods can be improved by increasing the sample count.

7.1.2 Other Layered Depth Map Methods

VPL-Path Tracing hybrid The VPL-path tracing hybrid [Tokuyoshi and Ogaki 2012b] has many parallels to our approach. One of the subtle differences is that they propose to use unsorted lists (unsorted depth values sequences). This implies that each depth values sequence must be searched linearly from start to end in order to find an intersection. As such, our intersection querying method should be faster because the search can be terminated early (due to the L_p sequences being sorted). The more interesting question is whether the time we spend pre-sorting the linked lists is actually amortized by the faster intersection queries. That is, it might be faster to use a more naive, unsorted construction mechanism (as [Tokuyoshi and Ogaki 2012b] proposes) and then search the lists in full length. Our results definitely back this claim: List construction far outweighs photon tracing on average. Even if this is indeed the case, it would only be a small optimization in our use case. The photon splatting step is still a huge factor in the overall frame time. Lastly, it should be noted that the PSPPSLL

were not documented in the literature at the time [Tokuyoshi and Ogaki 2012b] proposed their method.

Being based on VPLs, this hybrid method inherits some of VPLs' negative properties. Specifically, the bias due to bounding G. Furthermore, that light sources must be point-based because the VPLs are generated with reflective shadow maps.

An interesting idea brought up by the same authors is to use the imperfect shadow map technique to reduce layered depth map construction costs [Tokuyoshi and Ogaki 2012a]. It would be interesting to see the practical effects (in terms of performance and image quality) when applied to our method.

Ray-Marching We propose to always trace in the direction in which the layered depth map is oriented. This requires a large number of layered depth maps to cover the hemisphere adequately. Other authors propose to ray-march through fewer layered depth maps instead [Lischinski et al. 1998; Bürger et al. 2007; Niessner et al. 2010; Hu et al. 2014]. In fact, this has been the dominant approach thus far. The most significant benefit to ray-marching is that arbitrary directions can be used. This is why the ray-marching approach can be used to implement path tracing [Hu et al. 2014]. The problem is that the intersection tests are more expensive because of pixel crossings [Niessner et al. 2010]. Recall that [Hu et al. 2014] proposed to use a coarse voxel grid to mitigate this issue. This results in a very performant implementation. Specifically, [Hu et al. 2014] renders the sponza with path tracing in 4030–5110 ms depending on the view angle. Note that these measurements are very close to our results. However, path tracing is of course an unbiased method and as such produces more correct results. On the other hand, said measurements are based on a progressive path tracer. That is, they construct the layered depth maps once and re-use them for subsequent images. If we did the same, our approach would be significantly faster. Still, the splatting step would dominate the frame time.

It is difficult to asses which of the two approaches (fixed-direction or ray-marching) is most performant. From the timings, it seems that both approaches are in the same order of magnitude. We can only hypothesize about specific timings. In general, ray-marching fewer layered depth maps seems to be a better approach when the construction is the bottleneck and sampling is not. This is actually true for our indirect lighting method. Therefore, it would be interesting to test whether ray-marching would improve performance in our case (modulo splatting).

7.2 Method Improvements

In this section, we will list extensions and improvements to our proposed indirect lighting method. First, a multi-bounce extension is proposed. Second, a method to implement caustics. Third, a generalization to arbitrary light sources. Fourth, we propose to render progressively. Lastly, we suggest an improvement to our AO method.

7.2.1 Multi-bounce

Our method can be trivially extended to support multiple light bounces. As it is now, we split the primary photon into multiple secondary photons. The very same approach can be used to split the secondary photons into tertiary photons and so on.

The first practical difficulty is to stop the recursion. We propose to either fix the light bounces globally (like we did with one bounce) or to use Russian roulette. The problem with the latter is that it introduces temporal flickering. The second practical difficulty is to control the splitting. A naive extension of our approach would split photons exponentially based on the number of light bounces. Alternatively, one can choose a subset of the sample directions for the secondary bounces. This subset can then be reduced further for the tertiary bounces and so on. This scheme can be used to balance out the exponential growth. Another alternative is to choose sample directions randomly. Again, we chose not to do so for temporal coherence.

It should be noted that there is another implication with multiple bounces. Currently, the light's G-buffer contains all the needed information to split the primary photon into secondary photons. However, the secondary photons may leave the light's view. For such photons, the G-buffer cannot be queried for scene information. Instead, the layered depth maps must be augmented with the necessary scene information (surface BRDF and normal). This significantly increases the memory usage of the layered depth maps[18]. In our tests, we observed that the current memory requirements were already borderline acceptable. Specifically, the GTX 480 simply did not have enough memory for some configurations.

Assuming that the augmented layered depth maps fit in memory, then the above approach can be used to trace an arbitrary number of light bounces. Moreover, the tracing step itself is currently not a bottleneck. Thus the additional tracing overhead can be easily afforded. This is unlike other real-time multi-bounce approaches such as imperfect reflective shadow maps. The latter requires the scene to be rendered multiple times for secondary bounces which is very costly. In our proposed approach, the number of bounces is independent of the scene complexity.

Lastly, we note that the random direction approach could also be directly applied to the AO method. Again, the same bias–variance trade-off applies.

7.2.2 Non-Lambertian BRDFs

The problem with our current approach is that the outgoing directions are fixed. Therefore, perfect specular reflections are impossible. An approximate solution is to instead choose the outgoing direction which is closest to the perfect specular reflection. For large N, the difference would be negligible. The same approach can be used to implement glossy reflections.

In principle, any BRDF can be approximated this way. Of course, the quality of the result heavily depends on how well the hemisphere is sampled. That is, whether a truly representative direction can be chosen. Assuming this is possible, then our auxiliary data structure could in principle also be used to implement path tracing. Though in this extreme, it is arguably more practical to use one of the aforementioned ray-marching approaches.

7.2.3 Arbitrary Light Sources

Currently, photon emission is restricted to point light sources. This is so that the photon differentials can be easily traced via basic ray differential theory. Recall that ray differentials have been generalized to path differentials [Suykens and Willems 2009]. In turn, this theory can be used to emit photon differentials from arbitrary light sources [Frisvad et al. 2014]. Beyond the emission, the tracing itself is identical to tracing ray differentials. Therefore, the **Photon Tracing** step can be replaced with a compute shader that emits and traces photon differentials from arbitrary light sources. The **Photon**

[18] Diffuse reflectance can be encoded in two channels to reduce storage requirements [Kluczek 2014]. Likewise, the normals can be encoded. Further compaction specific to layered depth maps is also possible [Kerzner et al. 2013].

Splatting step would not have to be changed. Note that this not only enables area light sources such as disks and squares but also arbitrary geometry light sources. As mentioned earlier, the tracing step is currently not a bottleneck. Therefore, a more complex compute shader can easily be afforded.

7.2.4 Progressive Rendering

The scene can be rendered over multiple frames to improve visual quality (inspired by [Hu et al. 2014]). This is mostly relevant for interactive purposes. A simple approach is to randomly rotate the layered depth maps each frame so that new directions are sampled. The result is then averaged over several frames to produce a more convincing image. This is similar to the approach used by many path tracers. Of course, the random rotation would introduce noise in the result and temporal coherence is lost.

7.2.5 Screen-space AO Hybrid

Recall that one of the problem with our AO method was the lack of small-scale details. In contrast, such features are represented will in HBAO. Symmetrically, HBAO fails to capture large-scale details. Thus a combination of the two is an optimal solution. A similar principle is actually suggested in one of the first treatments of SSAO [Shanmugam and Arikan 2007]. HBAO is already a very fast method and even more so if it's limited to a small d_{max}. Thus the combination of HBAO and our approach could be performant in practice.

7.3 Implementation Improvements

In this section, we list improvements to the implementation. This will mostly be a technical discussion. First, we discuss vendor specific hardware extensions. Second, we suggest a bucket scheme to improve trace time. Third, we detail how memory use can be reduced.

7.3.1 Hardware Extensions

The implementations presented previously have not made use of any vendor specific hardware extensions. In the following, we list some of these extensions and how they can be used to improve the implementation. Of course, using any of these would restrict the implementation to a subset of GPUs.

Critical Sections in fragment shaders Recall the critical section discussion in Section 3.2.17. All of the extensions which provide critical sections in fragment shaders are vendor specific. While we believe that it is more performant to implement pre-sorting using fine-grained atomic operations, it would be interesting to see if a simple critical section actually outperformed our efforts. Theoretically, it would be counter-intuitive. Practically, the critical sections may have access to hardware features which we do not and thus benefit from hidden performance. We can only hypothesize without an actual implementation. Profiling is needed to find the answer.

Raster Ordered View The Nvidia GeForce GTX 980 comes with a DirectX feature called raster ordered view [Nvidia 2014]. The description is vague but this feature is supposedly intended to produce OIT. It remains to be seen if this is a new feature or simply the DirectX equivalent of Nvidia's critical section extension (NV_fragment_shader_interlock) [Brown et al. 2014b]. If it is indeed a new feature, then it could potentially be used to construct layered depth maps.

Viewport Multicast This Nvidia extension (nv_viewport_array2) allows the scene to be rasterized to multiple viewports at a time [Bolz et al. 2014a]. This feature could potentially be exploited to construct multiple layered depth maps in a single pass. Currently, each layered depth maps is constructed in its own pass with all the overhead that ensues. By batching the construction, this overhead can be mitigated.

Note that batching via the geometry shader is already possible [Bürger et al. 2007]. The approach is equivalent to MRT for the fragment shader but instead applied to the geometry shader.

Pointers The Nvidia-specific SSBO extension exposes direct access to the underlying buffer via pointers [Brown 2012]. Semantically, this is equivalent to the standard array-like accessing scheme. Still, it would be interesting to see if pointers performed better. E.g., due to compile-time optimizations or simply because of vendor specific optimizations of pointer indirection.

7.3.2 Buckets (Depth Ranges)

The L_p sequence can be split into buckets according to depth intervals. I.e., by dividing the singly linked list into an array of smaller singly linked lists. This could speed up tracing. First, the outer array is traversed until the right bucket is found. Then, the singly linked list in that bucket is traversed to find the exact intersection. This approach is inspired by the bucket sort method [Liu et al. 2009b] and depth ranges [Vasilakis and Fudos 2013]. Note that this is the opposite of PPPSLL (which is a singly linked list of arrays).

7.3.3 Improving Memory Usage

In the current implementation, the list node structure (depth value and next index) is also used to store the head indices. This has been done to simplify the implementation. Thus a lot of memory is wasted on storing a non-existent depth value next to each head index. Ideally, the head indices should just be stored contiguously next to each other.

7.4 Auxiliary Uses

In this section, we list some auxiliary uses of layered depth maps. When the layered depth maps are used for multiple purposes, their construction cost is amortized. Such auxiliary uses include:

- **Metaballs.** Metaball rendering can be accelerated using layered depth maps [Szécsi and Illés 2012].
- **Constructive Solid Geometry.** Presentations on OIT are often accompanied by how layered depth maps can be used to render constructive solid geometry [Lefebvre et al. 2014].
- **Linked Lists of Lights.** Per-pixel linked lists of lights have recently been proposed to speed up multi-light rendering [Abdul 2014]. In principle, such a light list could be combined with a layered depth maps from the user's view.
- **Molecular Surfaces.** A combination of OIT and constructive solid geometry to render molecular objects [Kauker et al. 2013].
- **Dynamics.** Like rendering, simulation of dynamics also requires ray-tracing. E.g., rigid body simulation in a physics engine.

This is be no means an exhaustive list. Please refer to [Vasilakis and Fudos 2014] and [Knowles et al. 2014] for more suggestions.

8 Conclusion

We have presented two global illumination methods using an auxiliary data structure based on layered depth maps. Said data structure lived up to the requirements which we defined in Section 3.3.2. In short, the data structure can be used to query global scene information in rasterization. Specifically, we presented a real-time AO method and an interactive single-bounce indirect lighting method which both convincingly resembles the path traced reference. Moreover, we presented a combination of the two methods to simulate both direct and indirect lighting from a point source together with environment lighting. The indirect lighting method is novel in the way it handles the diffuse reflection of photon differentials. The pre-sorted layered depth maps allows us to quickly find the first occluder in both directions with a novel trace algorithm. Lastly, our comparative study of layered depth map (Section 3.2) can also be used as a reference on OIT methods.

The indirect lighting method can be trivially extended to also store the direct photons. The quality of the photon differentials makes this a viable approach. Thus our work on indirect lighting could potentially be extended to handle both direct and indirect lighting in a unified approach. As mentioned earlier, the approach can also be extended with multiple light bounces and arbitrary light sources. There is also a lot of work to be done on non-Lambertian BRDFs. This is definitely an interesting direction for future study.

The layered depth maps are constructed independently of one another. In that sense, our auxiliary data structure is actually a collection of individual data structures that each stores a scene representation. Contrast this to, say, a voxel grid which stores a scene representation in a single data structure. The problem with a collection of individual data structures is that scene information may be duplicated. That is, the same surface point may be stored in multiple layered depth maps. This wastes space. There has been research on optimizing the data storage by sharing information between list nodes [Kerzner et al. 2013]. It would be interesting to see if entire list nodes could be shared between the layered depth maps in order to reduce data duplication.

Another proposal for future work is to test other layered depth map implementations. E.g., the l-buffer or HA-buffer. Unfortunately, this process is complicated by driver issues as we experienced ourselves. This makes the implementation process needlessly complex. Furthermore, we found that it was seldom the construction step which dominated layered depth map construction. Still, a practical study of layered depth map implementations would be useful for future reference.

Lastly, we would like to point out that our real-time and interactive methods can of course also be used in an offline context. We have strived to stay physically correct and only introduced bias to gain reasonable performance. The bias can be reduced by increasing the sampling density. We envision that our layered depth map approach can ultimately be used as a unified lighting solution.

Glossary

z-buffer 4, 5, 70, see *depth map*
z-value 4, 8, 10, see *depth value*

depth buffer 4, 12, 17, see *depth map*
depth map Map that contains per-pixel depth values. The depth map is a special case of a layered depth map. 1, 3–6, 29, 30, 35, 42, 43, 48, 52, 53, see *depth value* & *layered depth map*
depth value The distance from the viwer into the screen. Usually, the view is oriented so that the negative z-axis extends into the screen in EC. As such, the depth value is the negated z-coordinate in EC. However, depth values are not limited to EC and may also be extracted from later coordinate spaces. E.g., CC, NDC, SC, etc. Consequently, depth values are not necessarily linear. 2–10, 12, 13, 15–17, 19, 20, 24, 25, 29, 30, 34, 42, 43, 48, 66, 68, 70, 71

fragment lists 5, see *layered depth map*
fragment shader "Fragment shaders affect the processing of fragments during rasterization." as stated in the OpenGL 4.5 specification [Segal et al. 2014]. Note that a fragment shader may be invoked multiple times for the same pixel. E.g., due to overlapping geometry. 2, 3, 13, 19–23, 35, 42, 46, 47, 56, 68, 70, see *shader*
framebuffer "The framebuffer consists of a set of pixels arranged as a two-dimensional array. [...] each pixel in the framebuffer is simply a set of some number of bits." as stated in the OpenGL 4.5 specification [Segal et al. 2014]. 6, 7, 20, 25, 43, 46, 49, 50, 70

layered depth images 5, see *layered depth map*
layered depth map Map that may contain multiple depth values per pixel. 1–9, 13–17, 19, 22, 24, 25, 27, 29–32, 34–38, 42, 43, 45–47, 52–56, 59, 63, 64, 66–70, see *depth value*
layered fragment buffer 5, see *layered depth map*

multi-layer z-buffer 5, see *layered depth map*

shader "A shader specifies operations that are meant to occur on data as it moves through different programmable stages of the OpenGL processing pipeline, starting with vertices specified by the application and ending with fragments prior to being written to the framebuffer. The programming language used for shaders is described in the OpenGL Shading Language Specification." as stated in the OpenGL 4.5 specification [Segal et al. 2014]. 1, 2, 7, 9, 17, 19, 22, 24, 49, 56, 70
shader invocation The execution of a shader's *main* function. A vertex shader is invoked for every incoming vertex. A geometry shader is invoked for every incoming primitive. A fragment shader is invoked for every incoming fragment. 7–9, 19, 20, 22, see *shader* & *fragment shader*

Acronyms

Computer Science
- **FIFO** first-in, first-out. 6
- **RMW** read-modify-write. 7, 9, 10, 13, 56

Coordinate Systems
- **CC** clip coordinates. 50, 70
- **EC** eye coordinates. 48, 70
- **FC** filter coordinates. 41
- **NDC** normalized device coordinates. 4, 16, 24, 40, 70
- **SC** screen coordinates. 26, 70
- **TC** texture coordinates. 47
- **WC** world coordinates. 15, 16, 24, 26, 34, 40–42, 46, 48

Layered Depth Map Implementations
- Z^3 6–8, 10, 13, 14, 17
- **k^+-buffer** 13, 14, 17
- **k-buffer** 7, 9, 13, 14, 17, 42
- **A-buffer** anti-aliased, area-averaged, accumulation buffer. 5, 6, 9, 10, 12, 14, 17, 70
- **D-buffer** dequeue buffer. 11, 12, 14
- **DF-buffer** dynamic fragment buffer. 10–12, 14, 17
- **F-buffer** fragment-stream buffer. 6, 9, 14, 17
- **HA-buffer** hashed A-buffer. 12, 14, 17, 18, 24, 69
- **l-buffer** layered buffer or list buffer. 10–12, 14, 17, 69
- **PPFLA** per-pixel fixed-length arrays. 8, 11, 14, 17
- **PPPSLL** paged per-pixel singly linked lists. 9, 14, 17, 68
- **PPSLL** per-pixel singly linked lists. 9–12, 14, 17, 19–21, 24, 42, 43
- **PSPPFLA** pre-sorted per-pixel fixed-length arrays. 8, 12–14, 17
- **PSPPSLL** pre-sorted per-pixel singly linked lists. 12, 14, 17–20, 23, 24, 43, 66
- **PSPPVLA** pre-sorted per-pixel variable-length arrays. 12, 14, 17
- **R-buffer** recirculating fragment buffer. 6, 9, 14, 17
- **S-buffer** sparsity-aware buffer. 11, 12, 14, 17

OpenGL
- **GLSL** OpenGL shading language. 9
- **MRT** multiple render targets. 7, 68
- **RT** render target. 6–8, 18
- **SSBO** shader storage buffer object. 7–10, 12, 17, 19, 20, 24, 25, 46, 49, 50, 66, 68

Rendering
- **AO** ambient obscurance. 1, 28–38, 42, 48, 52, 53, 56, 59, 63, 67–69
- **AO** ambient occlusion. 1–3, 27, 28, 30, 52, 53
- **BRDF** bidirectional reflectance distribution function. 38, 45, 46, 48, 50, 66, 67, 69
- **BSDF** bidirectional scattering distribution function. 37
- **HBAO** horizon-based ambient occlusion. 30, 35, 43, 52–55, 68, 76
- **OIT** order-independent transparency. 5–7, 13, 15, 17, 31, 68, 69
- **PDF** probability density function. 29, 32, 37
- **SSAO** screen-space ambient occlusion. 3, 29–31, 53, 68
- **VPL** virtual point light. 37, 38, 40, 42, 43, 66, 67

Symbols

Radiometry

f_r **bidirectional reflectance distribution function** $\left[\mathrm{sr}^{-1}\right]$ 27, 38, 66

f_s **bidirectional scattering distribution function** $\left[\mathrm{sr}^{-1}\right]$ 27

I **radiant intensity** $\left[\mathrm{W\,sr}^{-1}\right]$ 38

E **irradiance** $\left[\mathrm{W\,m}^{-2}\right]$ 39–41, 46, 49, 50, 66

L **radiance** $\left[\mathrm{W\,m}^{-2}\,\mathrm{sr}^{-1}\right]$ 27, 33, 38–42, 49, 50

Φ **radiant flux** $[\mathrm{W}]$ 38, 39, 41, 42, 45–47, 49

Rendering

L_p^k **sorted depth value sequence** A sorted sequence of k ascending depth values belonging to pixel p. Note that k may vary from pixel to pixel. See Section 3.1.2 for the formal definition. 4–8, 10, 13, 71, see *layered depth map*

L_p **sorted depth value sequence** k may be omitted from L_p^k to denote that the sequence is not of fixed length. 5, 6, 9, 10, 12, 15, 16, 24, 34, 42, 43, 52, 66, 68

References

AALUND, F., AND BÆRENTZEN, A., 2013. A comparative study of screen-space ambient occlusion methods. Technical University of Denmark, Bachelor Thesis.

ABDUL, B., 2014. Real-time lighting via light linked list.

AILA, T., MIETTINEN, V., AND NORDLUND, P. 2003. Delay streams for graphics hardware. In *ACM SIGGRAPH 2003 Papers*, ACM, New York, NY, USA, SIGGRAPH '03, 792–800.

AKENINE-MÖLLER, T., HAINES, E., AND HOFFMAN, N. 2008. Real-time rendering 3rd edition. 1–27, 134–141.

BAKER, M. J., 2015. Maths - transformations using quaternions.

BAUER, F., KNUTH, M., AND BENDER, J. 2013. Screen-space ambient occlusion using a-buffer techniques. 140–147.

BAVOIL, L., AND MYERS, K. 2011. Order independent transparency with dual depth peeling.

BAVOIL, L., AND SAINZ, M. 2009. Multi-layer dual-resolution screen-space ambient occlusion. *SIGGRAPH 2009: Talks, SIGGRAPH '09*, 45.

BAVOIL, L., CALLAHAN, S. P., LEFOHN, A., COMBA, J. A. L. D., AND SILVA, C. T. 2007. Multi-fragment effects on the gpu using the k-buffer. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, USA, I3D '07, 97–104.

BAVOIL, L., SAINZ, M., AND DIMITROV, R. 2008. Image-space horizon-based ambient occlusion. *Siggraph'08: Acm Siggraph Talks 2008, Siggraph: Acm Siggraph Talks*.

BOLZ, J., BROWN, P., DODD, C., KILGARD, M., AND WERNESS, E., 2010. Nv_shader_buffer_load. Extension #379.

BOLZ, J., BROWN, P., AND HEYER, M., 2014. Nv_viewport_array2. Extension #476.

BOLZ, J., BROWN, P., LICHTENBELT, B., LICEA-KANE, B., WERNESS, E., SELLERS, G., ROTH, G., HAEMEL, N., BOUDIER, P., AND DANIELL, P., 2014. Arb_shader_image_load_store. Extension #115.

BRETON, J., BROWN, P., WERNESS, E., AND KILGARD, M., 2014. Nv_shader_thread_group. Extension #447.

BROWN, P., BOLZ, J., DANIELL, P., RICCIO, C., SELLERS, G., MERRY, B., AND KESSENICH, J., 2014. Arb_shader_storage_buffer_object. ARB Extension #137.

BROWN, P., BOLZ, J., AND HEYER, M., 2014. Nv_fragment_shader_interlock. Not yet assigned an extension number.

BROWN, P., 2012. Nv_shader_buffer_store. Extension #390.

BÜRGER, K., HERTEL, S., KRÜGER, J., AND WESTERMANN, R. 2007. Gpu rendering of secondary effects. In *Vision, Modeling and Visualization 2007*.

CALLAHAN, S., IKITS, M., COMBA, J., AND SILVA, C. 2005. Hardware-assisted visibility sorting for unstructured volume rendering. *IEEE TRANSACTIONS ON VISUALIZATION AND COMPUTER GRAPHICS 11*, 3, 285–295.

CARPENTER, L. 1984. The a-buffer, an antialiased hidden surface method. *Computers and Graphics 18*, 3, 103–108.

CATMULL, E. E. 1974. A subdivision algorithm for computer display of curved surfaces.

COLLOMB, C., 2007. A tutorial on inverting 3 by 3 matrices with cross products. Available online: http://www.emptyloop.com/technotes/.

COMBA, J. L. D., TORCHELSEN, R., BASTOS, R., AND MAULE, M. 2012. Memory-efficient order-independent transparency with dynamic fragment buffer. *Brazilian Symposium of Computer Graphic and Image Processing*, 134–141.

COOK, R. L., AND TORRANCE, K. E. 1982. A reflectance model for computer graphics. *ACM Transactions on Graphics 1*, 1, 7–24.

CRASSIN, C., NEYRET, F., SAINZ, M., GREEN, S., AND EISEMANN, E. 2011. Interactive indirect illumination using voxel cone tracing. *Symposium on Interactive 3D Graphics*, 207–207.

CRASSIN, C., 2010. Fast and accurate single-pass a-buffer using opengl 4.0+.

CRASSIN, C., 2010. Opengl 4.0+ abuffer v2.0: Linked lists of fragment pages.

DACHSBACHER, C., AND STAMMINGER, M. 2005. Reflective shadow maps. *Proceedings of the Symposium on Interactive 3d Graphics, Proc Symp Interactive 3d Graphics*, 203–208.

DACHSBACHER, C., KRIVANEK, J., HASAN, M., ARBREE, A., WALTER, B., AND NOVAK, J. 2014. Scalable realistic rendering with many-light methods. *COMPUTER GRAPHICS FORUM 33*, 1, 88–104.

DAVIES, L., 2014. Order-independent transparency approximation with pixel synchronization (update 2014).

DIMITROV, R., BAVOIL, L., AND SAINZ, M. 2008. Horizon-split ambient occlusion. In *Proceedings of the 2008 symposium on Interactive 3D graphics and games*, ACM, New York, NY, USA, I3D '08, 5:1–5:1.

DUTRÉ, P., PHILIPPE, B., AND BALA, K. 2006. *Advanced Global Illumination*. Peters.

ENDERTON, E., SINTORN, E., SHIRLEY, P., AND LUEBKE, D. 2011. Stochastic transparency. *Ieee Transactions on Visualization and Computer Graphics, Ieee Trans Visual Comput Graphics 17*, 8, 1036–1047.

EVERITT, C. 2001. Interactive order-independent transparency.

FABIANOWSKI, B., AND DINGLIANA, J. 2009. Interactive global photon mapping. *COMPUTER GRAPHICS FORUM 28*, 4, 1151–1159.

FILION, D., AND MCNAUGHTON, R. 2008. Effects & techniques. In *ACM SIGGRAPH 2008 Games*, ACM, New York, NY, USA, SIGGRAPH '08, 133–164.

FRISVAD, J. R., SCHJØTH, L., ERLEBEN, K., AND SPORRING, J. 2014. Photon differential splatting for rendering caustics. *Computer Graphics Forum 33*, 6, 252–263.

FRISVAD, J. R. 2012. Photon differentials: Adaptive anistropic density estimation in photon mapping. In *State of the Art in Photon Density Estimation*, vol. Article 6.

GORAL, C. M., TORRANCE, K. E., GREENBERG, D. P., AND BATTAILE, B. 1984. Modeling the interaction of light between diffuse surfaces. *Computer Graphics (ACM) 18*, 3, 213–222.

GORTLER, S. J., COHEN, M. F., AND HE, L.-W. 1997. Rendering layered depth images.

Govindaraju, N. K., Lin, M. C., and Manocha, D. 2004. Vissort: Fast visibility ordering of 3-d geometric primitives. Tech. rep., University of North Carolina at Chapel Hill.

Grajewski, S., Foley, T., Insko, B., Janczak, T., Salvi, M., Seiler, L., and Poniecki, T., 2013. Gl_intel_fragment_shader_ordering. OpenGL Extension #441.

Gruen, H., and Thibieroz, N. 2010. Oit and indirect illumination using dx11 linked lists. In *Proceedings of the 2010 Game Developers Conference*.

Hachisuka, T. 2005. High-quality global illumination rendering using rasterization. In *GPU Gems 2*, M. Pharr, Ed. Addison-Wesley, 615–633.

Harris, T. L. 2001. A pragmatic implementation of non-blocking linked-lists. In *Lecture Notes in Computer Science*, Springer-Verlag, 300–314.

Heckbert, P. S. 1990. Adaptive radiosity textures for bidirectional ray tracing. *Computer Graphics 24*, 4, 145–154.

Hermes, J., Henrich, N., Grosch, T., and Mueller, S. 2010. Global illumination using parallel global ray-bundles. *Vmv 2010 - Vision, Modeling and Visualization, Vmv - Vis., Model. Vis*, 65–72.

Holbert, D., 2011. Saying goodbyeẗo shadow acne.

Houston, M., Preetham, A. J., and Segal, M. 2005. Stanford tech report- cstr 2005-05 (2005), pp. 1–6 a hardware f-buffer implementation.

Hu, W., Huang, Y., Zhang, F., Yuan, G., and Li, W. 2014. Ray tracing via gpu rasterization. *VISUAL COMPUTER 30*, 6-8, 697–706.

Igehy, H. 1999. Tracing ray differentials. *SIGGRAPH 99 CONFERENCE PROCEEDINGS*, 179–186.

Jarosz, W., Jensen, H. W., and Donner, C. 2008. Advanced global illumination using photon mapping. In *ACM SIGGRAPH 2008 Classes*, ACM, New York, NY, USA, SIGGRAPH '08, 2:1–2:112.

JeGX, 2014. How to rotate a vertex by a quaternion in glsl.

Jensen, H. W., and Christensen, N. J. 1995. Photon maps in bidirectional monte carlo ray tracing of complex objects. *Computers and Graphics 19*, 2, 215–224.

Jouppi, N. P., and Chang, C.-F. 1999. Z3: an economical hardware technique for high-quality antialiasing and transparency. *SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, 85–93.

Kajiya, J. T. 1986. The rendering equation. *SIGGRAPH Comput. Graph. 20*, 4 (aug), 143–150.

Kasyan, N., Schulz, N., and Sousa, T., 2011. Secrets of cryengine 3 graphics technology. Slideshow accompanying a presentation.

Kauker, D., Krone, M., Panagiotidis, A., Reina, G., and Ertl, T. 2013. Rendering molecular surfaces using order-independent transparency. *Eurographics Workshop on Parallel Graphics and Visualization*, 33–40.

Keller, A. 1997. Instant radiosity. *Proceedings of the Acm Siggraph Conference on Computer Graphics, Proc Acm Siggraph Conf Computer Graph*, 49–54.

Kerzner, E., Wyman, C., Butler, L., and Gribble, C. 2013. Toward efficient and accurate order-independent transparency. *Acm Siggraph 2013 Posters, Siggraph 2013, Acm Siggraph Posters, Siggraph*.

Kessenich, J., LunarG, Baldwin, D., and Rost, R., 2014. The opengl shading language (version 4.5).

Kluczek, K. 2014. Reducing texture memory usage by 2-channel color encoding. In *GPU Pro 5: Advanced Rendering Techniques*, W. Engel, Ed. CRC Press, 25–32.

Knowles, P., Leach, G., and Zambetta, F. 2012. Efficient layered fragment buffer techniques. In *OpenGL Insights*, CRC Press, P. Cozzi and C. Riccio, Eds., 279–292.

Knowles, P., Leach, G., and Zambetta, F. 2013. Backwards Memory Allocation and Improved OIT. In *Proceedings of Pacific Graphics 2013 (short papers)*, 59–64.

Knowles, P., Leach, G., and Zambetta, F. 2014. Fast sorting for exact oit of complex scenes. *VISUAL COMPUTER 30*, 6-8, 603–613.

Krüger, J., Bürger, K., and Westermann, R. 2006. Interactive screen-space accurate photon tracing on gpus. In *Proceedings of the 17th Eurographics Conference on Rendering Techniques*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, EGSR '06, 319–329.

Kubisch, C. 2014. Order independent transparency in opengl 4.x. In *Proceedings of the 2014 GPU Technology Conference*.

Landis, H. 2002. Production-Ready Global Illumination. In *Siggraph Course Notes*, vol. 16.

Lee, J.-A., and Kim, L.-S. 2000. Single-pass full-screen hardware accelerated antialiasing. *Proceedings of the SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 67–75.

Lefebvre, S., Hornus, S., and Lasram, A. 2013. Ha-buffer: Coherent hashing for single-pass a-buffer. Rapport de recherche RR-8282, INRIA, Apr.

Lefebvre, S., Hornus, S., and Lasram, A. 2014. Per-pixel lists for single pass a-buffer. In *GPU Pro 5: Advanced Rendering Techniques*, W. Engel, Ed. CRC Press, 3–23.

Leopardi, P. 2006. A partition of the unit sphere into regions of equal area and small diameter. *ELECTRONIC TRANSACTIONS ON NUMERICAL ANALYSIS 25*, 309–327.

Licea-Kane, B., Lichtenbelt, B., Dodd, C., Werness, E., Sellers, G., Roth, G., Bolz, J., Haemel, N., Brown, P., Boudier, P., and Daniell, P., 2012. Arb_shader_atomic_counters. ARB Extension #114.

Lipowski, J. K. 2010. Multi-layered framebuffer condensation: The l-buffer concept. *Computer Vision And Graphics, Part 2 6375*, 89–97.

Lipowski, J. K. 2013. D-buffer: irregular image data storage made practical. *OPTO-ELECTRONICS REVIEW 21*, 1, 103–125.

Lischinski, D., Rappoport, A., Drettakis, G., and Max, N. 1998. Image-based rendering for non-diffuse synthetic scenes.

Liu, B., Wei, L.-Y., and Xu, Y.-Q. 2006. Multi-layer depth peeling via fragment sort. Tech. Rep. MSR-TR-2006-81, Microsoft Research, June.

Liu, F., Huang, M.-C., Liu, X.-H., and Wu, E.-H. 2009. Single pass depth peeling via cuda rasterizer. *Siggraph 2009: Talks, Siggraph '09, Siggraph : Talks, Siggraph*.

Liu, F., Huang, M.-C., Liu, X.-H., Wu, E.-H., and Wu, E.-H. 2009. Efficient depth peeling via bucket sort. *Proceedings of the SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 51–57.

Liu, F., Huang, M.-C., Liu, X.-H., and Wu, E.-H. 2010. Freepipe: A programmable parallel rendering architecture for efficient multi-fragment effects. *Proceedings of I3D 2010: the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 75–82.

Liu, F., Liu, F., Liu, X., Song, Y., and Xu, X. 2013. Multi-layer screen-space ambient occlusion using hybrid sampling. *Proceedings - Vrcai 2013: 12th Acm Siggraph International Conference on Virtual-reality Continuum and Its Applications in Industry, Proc. - Vrcai: Acm Siggraph Int. Conf. Virtual-real. Continuum Its Appl. Ind*, 71–75.

Loos, B. J., and Sloan, P.-P. 2010. Volumetric obscurance. In *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, USA, I3D '10, 151–156.

Mammen, A. 1989. Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. *IEEE Computer Graphics And Applications 9*, 4, 43–55.

Mark, W., and Proudfoot, K. 2001. The f-buffer: A rasterization-order fifo buffer for multi-pass rendering. *Proceedings of the ACM SIGGRAPH Conference on Computer Graphics*, WORKSHOP, 57–63.

Maule, M., Comba, J. L. D., Torchelsen, R. P., and Bastos, R. 2011. A survey of raster-based transparency techniques. *COMPUTERS and GRAPHICS-UK 35*, 6, 1023–1034.

Maule, M., Comba, J. a., Torchelsen, R., and Bastos, R. 2013. Hybrid transparency. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, USA, I3D '13, 103–118.

Max, N., Pueyo, X., and Schroder, P. 1996. Hierarchical rendering of trees from precomputed multi-layer z-buffers.

McDonald, J., 2013. Alpha blending: To pre or not to pre.

McGuire, M., Osman, B., Bukowski, M., and Hennessy, P. 2011. The alchemy screen-space ambient obscurance algorithm. *Proceedings - Hpg 2011: Acm Siggraph Symposium on High Performance Graphics, Proc. - Hpg: Acm Siggraph Symp. High Perform. Graph*, 25–32.

Meshkin, H., 2007. Prefix sum pass to linearize a-buffer storage.

Mittring, M. 2007. Finding next gen: Cryengine 2 (course notes). In *ACM SIGGRAPH 2007 courses*, ACM, New York, NY, USA, SIGGRAPH '07, 97–121.

Mittring, M., 2012. The technology behind the unreal engine 4 elemental demo. Slideshow accompanying a presentation.

Myers, K., and Bavoil, L. 2007. Stencil routed a-buffer. In *ACM SIGGRAPH 2007 Sketches*, ACM, New York, NY, USA, SIGGRAPH '07.

Nalbach, O., Ritschel, T., and Seidel, H.-P. 2014. Deep screen space. In *Proceedings of the 18th Meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, USA, I3D '14, 79–86.

Nichols, G., and Wyman, C. 2009. Multiresolution splatting for indirect illumination. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, USA, I3D '09, 83–90.

Niessner, M., Schaefer, H., and Stamminger, M. 2010. Fast indirect illumination using layered depth images. *VISUAL COMPUTER 26*, 6–8, 679–686.

Nvidia, 2014. Nvidia geforce gtx 980.

Patney, A., Tzeng, S., and Owens, J. D. 2010. Fragment-parallel composite and filter. *COMPUTER GRAPHICS FORUM 29*, 4, 1251–1258.

Peeper, C., 2008. Prefix sum pass to linearize a-buffer storage. US Patent App. 11/766,091.

Pharr, M., and Humphreys, G. 2004. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Porter, T., and Duff, T. 1984. Compositing digital images. *Computers and Graphics 18*, 3, 253–259.

Ren, Z., Hua, W., Chen, L., and Bao, H. 2005. Intersection fields for interactive global illumination. *VISUAL COMPUTER 21*, 8-10, 569–578.

Riccio, C., 2015. Opengl hardware extension matrix.

Ritschel, T., Grosch, T., Kim, M. H., Seidel, H.-P., Dachsbacher, C., and Kautz, J. 2008. Imperfect shadow maps for efficient computation of indirect illumination. *ACM Transactions on Graphics 27*, 5, 129.

Ritschel, T., Grosch, T., and Seidel, H. P. 2009. Approximating dynamic global illumination in image space. *Proceedings of I3d 2009: the 2009 Acm Siggraph Symposium on Interactive 3d Graphics and Games, Proc. I3d: Acm Siggraph Symp. Interact. 3d Graph. Games*, 75–82.

Salvi, M., and Vaidyanathan, K. 2014. Multi-layer alpha blending. *Symposium on Interactive 3D Graphics*, 151–158.

Salvi, M., Lefohnz, A., and Montgomery, J. 2011. Adaptive transparency. *Proceedings - Hpg 2011: Acm Siggraph Symposium on High Performance Graphics, Proc. - Hpg: Acm Siggraph Symp. High Perform. Graph*, 119–126.

Salvi, M. 2013. Pixel synchronization: Solving old graphics problems with new data structures. In *ACM SIGGRAPH 2013 courses, Advances in Real-time Rendering in Games*, ACM, New York, NY, USA, SIGGRAPH '13'.

Sam, 2014. Beautiful maths simplification: quaternion from two vectors.

Sbert, M., and Sàndez, X. 1996. *The use of global random directions to compute radiosity. Global Monte Carlo techniques*.

Schilling, A., and StraBer, W. 1993. Exact: Algorithm and hardware architecture for an improved a-buffer. *Proc ACM SIGGRAPH 93 Conf Comput Graphics*, 85–91.

Schjøth, L., Frisvad, J. R., Erleben, K., and Sporring, J. 2007. Photon differentials. *Proceedings of Graphite 2007*, 179–186.

Segal, M., Akeley, K., Frazier, C., Leech, J., and Brown, P., 2013. The opengl graphics system: A specification (version 4.3 core profile).

Segal, M., Akeley, K., Frazier, C., Leech, J., and Brown, P., 2014. The opengl graphics system: A specification (version 4.5 core profile).

Shanmugam, P., and Arikan, O. 2007. Hardware accelerated ambient occlusion techniques on gpus. *I3D 2007: ACM SIGGRAPH SYMPOSIUM ON INTERACTIVE 3D GRAPHICS AND GAMES, PROCEEDINGS*, 73–80.

Sugihara, M., Rauwendaal, R., and Salvi, M. 2014. Layered reflective shadow maps for voxel-based indirect illumination. 117–125.

Suykens, F., and Willems, Y. D. 2009. Path differentials and applications.

Szécsi, L., and Illés, D. 2012. Real-time metaball ray casting with fragment lists. In *Eurographics (Short Papers) '12*, 93–96.

Thibieroz, N. 2011. Order-independent transparency using per-pixel linked lists. In *GPU Pro 2: Advanced Rendering Techniques*, A K Peters, W. Engel, Ed., 409–431.

Tokuyoshi, Y., and Ogaki, S., 2012. Imperfect ray-bundle tracing for interactive multi-bounce global illumination. High Performance Graphics 2012 Posters.

Tokuyoshi, Y., and Ogaki, S. 2012. Real-time bidirectional path tracing via rasterization. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, USA, I3D '12, 183–190.

Tokuyoshi, Y., Sekiney, T., and Ogakiz, S. 2011. Fast global illumination baking via ray-bundles. *SIGGRAPH Asia 2011 Sketches, SA'11*, 25.

Tokuyoshi, Y., Sekine, T., Da Silva, T., and Kanai, T. 2013. Adaptive ray-bundle tracing with memory usage prediction: Efficient global illumination in large scenes. *Computer Graphics Forum, Comput Graphics Forum 32*, 7, 315–324.

Vardis, K., Papaioannou, G., and Gaitatzes, A. 2013. Multi-view ambient occlusion with importance sampling. *Proceedings of the Symposium on Interactive 3d Graphics, Proc Symp Interactive 3d Graphics*, 111–118.

Vasilakis, A., and Fudos, I. 2012. S-buffer: Sparsity-aware multifragment rendering. In *Eurographics 2012 - Short Papers Proceedings*, 101–104.

Vasilakis, A. A., and Fudos, I. 2013. Depth-fighting aware methods for multifragment rendering. *Ieee Transactions on Visualization and Computer Graphics, Ieee Trans Visual Comput Graphics 19*, 6, 967–977.

Vasilakis, A. A., and Fudos, I. 2014. K+-buffer: Fragment synchronized k-buffer. In *Proceedings of the 18th Meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, USA, I3D '14, 143–150.

Veach, E., Pueyo, X., and Schroder, P. 1996. Non-symmetric scattering in light transport algorithms.

Wald, I., Kollig, T., Benthin, C., Keller, A., and Slusallek, P. 2002. Interactive global illumination using fast ray tracing. *Eurographics Workshop on Rendering*, 15–24.

Whitted, T. 1980. An improved illumination model for shaded display. *Commun. ACM 23*, 6 (June), 343–349.

Winner, S., Kelley, M., Pease, B., Rivard, B., and Yen, A. 1997. Hardware accelerated rendering of antialiasing using a modified a-buffer algorithm.

Wittenbrink, C. 2001. R-buffer: A pointerless a-buffer hardware architecture. *Proceedings of the ACM SIGGRAPH Conference on Computer Graphics*, WORKSHOP, 73–80.

Yang, J., and McKee, J. 2010. Real-time order independent transparency and indirect illumination using direct3d 11. In *Proceedings of the Siggraph 2010 course Advances in Real-Time Rendering in 3D Graphics and Games*.

Yang, J. C., Hensley, J., Gruen, H., and Thibieroz, N. 2010. Real-time concurrent linked list construction on the gpu. *COMPUTER GRAPHICS FORUM 29*, 4, 1297–1304.

Zhang, C., Hsieh, H.-H., and Shen, H.-W. 2008. Real-time reflections on curved objects using layered depth textures. *MCCSIS'08 - IADIS Multi Conference on Computer Science and Information Systems; Proceedings of Computer Graphics and Visualization 2008 and Gaming 2008: Design for Engaging Experience Soc. Interaction*, 276–281.

Zhukov, S., Iones, A., Kronin, G., Drettakis, G., and Max, N. 1998. An ambient light illumination model.

Zirr, T., Rehfeld, H., and Dachsbacher, C. 2014. Object-order ray tracing for fully dynamic scenes. In *GPU Pro 5: Advanced Rendering Techniques*, W. Engel, Ed. CRC Press, 419–438.

Appendix

Equality of Double Inversion used in HBAO

We will now prove the equality claimed in Section 4.2.3. That is, we want to prove that

$$\frac{1}{\pi}\int_{\mathcal{H}} F(\omega)\cos\theta \mathrm{d}\omega = 1 - \frac{1}{\pi}\int_{\mathcal{H}} (1 - F(\omega))\cos\theta \mathrm{d}\omega$$

where F is any function for which the integral is defined. $\mathcal{H}$ is the unit hemisphere defined by the normal, n. θ is the angle between ω and n. The equality can be shown by expanding the terms on the right-hand side

$$\begin{aligned}
1 - \frac{1}{\pi}\int_{\mathcal{H}} (1 - F(\omega))\cos\theta \mathrm{d}\omega &= 1 - \frac{1}{\pi}\left(\int_{\mathcal{H}} \cos\theta \mathrm{d}\omega - \int_{\mathcal{H}} F(\omega)\cos\theta \mathrm{d}\omega\right) \\
&= 1 - \frac{1}{\pi}\left(\int_{\phi=0}^{2\pi}\int_{\theta=0}^{\frac{\pi}{2}} \cos\theta\sin\theta \mathrm{d}\theta \mathrm{d}\phi - \int_{\mathcal{H}} F(\omega)\cos\theta \mathrm{d}\omega\right) \\
&= 1 - \frac{1}{\pi}\left(\pi - \int_{\mathcal{H}} F(\omega)\cos\theta \mathrm{d}\omega\right) \\
&= \frac{1}{\pi}\int_{\mathcal{H}} F(\omega)\cos\theta \mathrm{d}\omega
\end{aligned}$$

where the integral over the cosine term has been expanded using spherical coordinates.

Source Code

The full source code can be found in the Git repository at HTTPS://GITHUB.COM/FREDERIKAALUND/SFJ. Below, we have listed the subset of GLSL code which highlights the core functionality.

Listing 12: *Point Cloud Visualization*

```
#extension GL_NV_shader_buffer_load: enable
#extension GL_NV_gpu_shader5: enable
#extension GL_EXT_shader_image_load_store: enable

uniform samplerBuffer lights;
uniform ivec2 window_dimensions;



struct view_type {
    mat4 view_matrix, projection_matrix, view_projection_matrix;
    vec4 eye;
    vec4 right, forward, up;
    ivec2 dimensions;
};


layout(std140) uniform user_view_block
{ view_type user_view; };

uniform int ldm_view_count;

readonly restrict layout(std430) buffer view_block
{ view_type views[]; };

readonly restrict layout(std430) buffer data_offset_block
{ uvec4 data_offsets[]; };



struct ldm_data {
    uint32_t next;
    //uint32_t compressed_diffuse;
    float depth;
```

```
35 };
36 readonly restrict layout (std430) buffer data_buffer
37 { ldm_data data[]; };
38 writeonly restrict layout (std430) buffer debug_view_buffer
39 { uint32_t debug_view[]; };
40
41
42
43
44 void draw_layered_depth_map(
45     in vec2 ndc_position,
46     in view_type view,
47     in uint32_t data_offset,
48     in uint32_t id )
49 {
50     uint32_t heads_index = data_offset + uint32_t(gl_FragCoord.x) + uint32_t(gl_FragCoord.y) *
       window_dimensions.x;
51     uint32_t current = data[heads_index].next;
52
53     // Constants
54     const int max_list_length = 2000;
55
56     // Loop over each point
57     int list_length = 0;
58     while (0 != current && list_length < max_list_length) {
59         float depth = data[current].depth;
60         current = data[current].next;
61         list_length++;
62
63         vec3 wc_eye_position = view.eye.xyz;
64         vec3 right = view.right.xyz;
65         vec3 forward = view.forward.xyz;
66         vec3 up = view.up.xyz;
67
68         // Othographic
69         const float right_scale = 2000.0;
70         const float top_scale = 2000.0;
71
72         vec3 direction = (
73             forward * depth
74             + right * right_scale * ndc_position.x
75             + up * top_scale * ndc_position.y);
76         vec3 wc_sample_position = wc_eye_position + direction;
77
78         // Project into user view
79         vec4 cc_sample_position = user_view.view_projection_matrix * vec4(wc_sample_position, 1.0);
80         if (cc_sample_position.x > cc_sample_position.w || cc_sample_position.x < -cc_sample_position.w
81             || cc_sample_position.y > cc_sample_position.w || cc_sample_position.y < -cc_sample_position.w
82             || cc_sample_position.z > cc_sample_position.w || cc_sample_position.z < -cc_sample_position.w)
83             continue;
84         vec3 ndc_sample_position = cc_sample_position.xyz / cc_sample_position.w;
85         vec2 tc_sample_position = (ndc_sample_position.xy + vec2(1.0)) * 0.5;
86         uvec2 sc_sample_position = uvec2(tc_sample_position * user_view.dimensions);
87
88         // Write to buffer (as if it was a texture)
89         uint32_t index = sc_sample_position.x + sc_sample_position.y * user_view.dimensions.x;
90         debug_view[index] = id;
91     }
92 }
93
94
95
96 void main() {
97     vec2 tc_position = vec2(gl_FragCoord) / window_dimensions;
98     vec2 ndc_position = tc_position * 2.0 - vec2(1.0);
99
```

```
    for (int i = 0; i < ldm_view_count; ++i)
        draw_layered_depth_map(ndc_position, views[i], data_offsets[i][0], i + 1);
}
```

Listing 12: *Point Cloud Visualization*

Listing 13: *Layered Depth Map Clear Pass*

```
#extension GL_NV_shader_buffer_load: enable
#extension GL_NV_gpu_shader5: enable
#extension GL_EXT_shader_image_load_store: enable

uniform ivec2 window_dimensions;
uniform int ldm_view_count;


struct ldm_data {
    uint32_t next;
    float depth;
};
writeonly restrict layout (std430) buffer data_buffer
{ ldm_data data[]; };

writeonly restrict layout (std430) buffer debug_view_buffer
{ uint32_t debug_view[]; };

struct color_data {
    uint32_t r, g, b;
};


struct vertex_data
{
    vec3 wc_view_ray_direction;
};
noperspective in vertex_data vertex;
layout(pixel_center_integer) in uvec2 gl_FragCoord;

void main()
{
    uint32_t index = gl_FragCoord.x + gl_FragCoord.y * window_dimensions.x;

    for (int i = 0; i < ldm_view_count; ++i)
        data[i*window_dimensions.x*window_dimensions.y + index].next = 0;

    debug_view[index] = 0;
}
```

Listing 13: *Layered Depth Map Clear Pass*

Listing 14: *Layered Depth Map Construction Pass*

```
#extension GL_NV_shader_buffer_load: enable
#extension GL_NV_gpu_shader5: enable
#extension GL_EXT_shader_image_load_store: enable

uniform sampler2D diffuse_texture;

uniform ivec2 window_dimensions;
uniform uint32_t total_data_offset;



struct ldm_data {
    uint32_t next;
    //uint32_t compressed_diffuse;
    float depth;
};
```

```
17
18 layout(binding = 0, offset = 0) uniform atomic_uint count;
19 coherent restrict layout(std430) buffer data_buffer
20 { ldm_data data[]; };
21
22
23
24 struct vertex_data
25 {
26     float negative_ec_position_z;
27     vec2 oc_texture_coordinate;
28 };
29 in vertex_data vertex;
30
31
32
33 uint32_t allocate() { return total_data_offset + atomicCounterIncrement(count); }
34
35
36 uint32_t compress( in vec4 color )
37 { return (uint32_t(color.x * 255.0) << 24u) + (uint32_t(color.y * 255.0) << 16u) + (uint32_t(color.z * 255.0)
       << 8u) + (uint32_t(0.1*255.0)); }
38
39
40
41 void main()
42 {
43     //vec2 tc_texture_coordinates = vec2(vertex.oc_texture_coordinate.x, 1.0 - vertex.oc_texture_coordinate.y);
44     //vec4 diffuse = texture(diffuse_texture, tc_texture_coordinates);
45
46     //uint32_t compressed_diffuse = compress(diffuse);
47     //float depth = gl_FragCoord.z;
48     float depth = vertex.negative_ec_position_z;
49
50     // Calculate indices
51     uint32_t head = total_data_offset + uint32_t(gl_FragCoord.x) + uint32_t(gl_FragCoord.y) * window_dimensions
       .x;
52     uint32_t new = allocate();
53
54     // Store fragment data in node
55     //data[new].compressed_diffuse = compressed_diffuse;
56     data[new].depth = depth;
57
58     // Start with the head node
59     uint32_t previous = head;
60     uint32_t current = data[head].next;
61
62     // Insert the new node while maintaining a sorted list.
63     // The algorithm finishes in a finite yet indeterminate number of steps.
64     // Indeterminate, since some steps may be repeated due to concurrent updates.
65     // Thus the total number of steps required for a single insertion
66     // is not known beforehand. However, finiteness guarantees
67     // that the algorithm terminates eventually. In other words,
68     // it is a lock-free algorithm (though not wait-free).
69
70     for (;;)
71     //const int max_iterations = 2048;
72     //for (int i = 0; i < max_iterations; ++i)
73         // We are either at the end of the list or just before a node of greater depth...
74         if (current == 0 || depth < data[current].depth) {
75             // ...so we attempt to insert the new node here. First,
76             // the new node is set to point to the current node. It is crucial
77             // that this change happens now since the next step makes
78             // the new node visible to other threads. That is, the new node must
79             // be in a complete state before becoming visible.
80             data[new].next = current;
```

```

            // Memory barrier omitted for added performance.

            // Then the previous node is atomically updated to point to new node
            // if the previous node still points to the current node.
            // Returns the original content of data[previous].next (regardless of the
            // result of the comparison).
            uint32_t previous_next = atomicCompSwap(data[previous].next, current, new);

            // The atomic update occurred...
            if (previous_next == current)
                // ...so we are done.
                break;
            // Another thread updated data[previous].next before us...
            else
                // ...so we continue from previous_next
                current = previous_next;
        // We are still searching for a place to insert the new node...
        } else {
            // ...so we advance to the next node in the list.
            previous = current;
            current = data[current].next;
            //current = atomicAdd(data[current].next, 0); // Atomic read
        }
}
```

Listing 14: *Layered Depth Map Construction Pass*

Listing 15: *HBAO*

```
#define USE_RANDOM_DIRECTION 0

const int poisson_disc_size = 128;
uniform vec2 poisson_disc[poisson_disc_size];

uniform sampler2D depths;
uniform sampler2D wc_normals;
uniform sampler2D random;

uniform vec3 wc_view_eye_position;
uniform float z_far;

uniform ivec2 window_dimensions;

uniform mat4 view_matrix;
uniform mat4 projection_matrix;
uniform mat4 view_projection_matrix;
uniform mat4 inverse_view_projection_matrix;



struct vertex_data
{
    vec3 wc_view_ray_direction;
};
noperspective in vertex_data vertex;

layout(location = 0) out vec4 ambient_occlusion;



////////////////////////////////////////////////////////////////////////////////
/// Utility Functions
////////////////////////////////////////////////////////////////////////////////
vec2 get_one_over_tan_half_fov( in mat4 projection_matrix )
{ return vec2(projection_matrix[0][0], projection_matrix[1][1]); }
```

```
38
39 float get_tc_z(
40     in sampler2D sampler,
41     in vec2 tc_position )
42 { return texture(sampler, tc_position).z; }
43
44 float get_tc_z(
45     in sampler2D sampler,
46     in vec2 tc_position,
47     in vec2 tc_offset )
48 { return get_tc_z(sampler, tc_position + tc_offset); }
49
50 float get_ec_z(
51     in float tc_z,
52     in mat4 projection_matrix )
53 { return projection_matrix[3][2] / (-2.0 * tc_z + 1.0 - projection_matrix[2][2]); }
54
55 float get_ec_z(
56     in sampler2D sampler,
57     in vec2 tc_position,
58     in mat4 projection_matrix)
59 { return get_ec_z(get_tc_z(sampler, tc_position), projection_matrix); }
60
61 vec3 get_ec_position( in vec2 tc_position, in float ec_position_z, in mat4 projection_matrix )
62 {
63     vec2 one_over_tan_half_fov = get_one_over_tan_half_fov(projection_matrix);
64     vec2 tan_half_fov = 1.0 / one_over_tan_half_fov;
65
66     vec2 ndc_position = tc_position * vec2(2.0) - vec2(1.0);
67     vec2 cc_position = ndc_position * -ec_position_z;
68     return vec3(tan_half_fov * cc_position, ec_position_z);
69 }
70
71 vec3 get_ec_position( in sampler2D sampler, in vec2 tc_position, in mat4 projection_matrix )
72 {
73     float ec_position_z = get_ec_z(sampler, tc_position, projection_matrix);
74     return get_ec_position(tc_position, ec_position_z, projection_matrix);
75 }
76
77 vec2 get_tc_length( in float ec_length, in float ec_position_z, in mat4 projection_matrix )
78 {
79     vec2 one_over_tan_half_fov = get_one_over_tan_half_fov(projection_matrix);
80     return 0.5 * ec_length * one_over_tan_half_fov / -ec_position_z;
81 }
82
83
84 ////////////////////////////////////////////////////////////////////////////////
85 /// Trigonometric Functions
86 ////////////////////////////////////////////////////////////////////////////////
87 float tan_to_sin( in float x )
88 {
89     return x * pow(x * x + 1.0, -0.5);
90 }
91
92
93 vec3 minimum_difference( in vec3 p, in vec3 p_right, in vec3 p_left )
94 {
95     vec3 v1 = p_right - p;
96     vec3 v2 = p - p_left;
97     return (dot(v1, v1) < dot(v2, v2)) ? v1 : v2;
98 }
99
100 vec3 tangent_eye_pos( in sampler2D sampler, in vec2 tc, in vec4 tangentPlane, in mat4 projection_matrix )
101 {
102     // view vector going through the surface point at tc
103     vec3 V = get_ec_position(sampler, tc, projection_matrix);
```

```
    float NdotV = dot(tangentPlane.xyz, V);
    // intersect with tangent plane except for silhouette edges
    if (NdotV < 0.0) V *= (tangentPlane.w / NdotV);
    return V;
}



void main()
{
    vec2 tc_position = gl_FragCoord.xy / window_dimensions;
    vec3 ec_position = get_ec_position(depths, tc_position, projection_matrix);
    vec3 wc_normal = texture(wc_normals, tc_position).xyz;
    vec3 ec_normal = transpose(inverse(mat3(view_matrix))) * wc_normal;

    ambient_occlusion.a = 0.0;

    const int base_samples = 0;
    const int min_samples = 512;
    const float ec_radius = 1280.0;
    const float ec_radius_squared = ec_radius * ec_radius;
    const float bias = 0.0;

    const int samples = min_samples;

    vec2 tc_radius = get_tc_length(ec_radius, ec_position.z, projection_matrix);
    vec2 sc_radius = tc_radius * window_dimensions;


    if (sc_radius.x < 1.0)
    {
        ambient_occlusion.a = 1.0;
        return;
    }

    // Stepping
    const int max_steps = 128; // 8
    int steps = min(int(sc_radius.x), max_steps);


    vec3 random_direction = texture(random, tc_position).xyz;
    random_direction = normalize(random_direction * 2.0 - 1.0);

    float angle_step = 2.0 * PI / float(samples);
    float uniform_distribution_random = texture(random, tc_position).x;
    float alpha = uniform_distribution_random * PI * 2.0;
    mat2 random_rotation = mat2(cos(alpha), sin(alpha), -sin(alpha), cos(alpha));

    vec3 bent_normal = vec3(0.0);

    vec2 depths_size = textureSize(depths, 0);
    vec2 depths_size_inversed = vec2(1.0) / depths_size;

    vec3 p_right, p_left, p_top, p_bottom;
    vec4 tangentPlane = vec4(ec_normal, dot(ec_position, ec_normal));
    p_right = tangent_eye_pos(depths, tc_position + vec2(depths_size_inversed.x, 0.0), tangentPlane,
     projection_matrix);
    p_left = tangent_eye_pos(depths, tc_position + vec2(-depths_size_inversed.x, 0.0), tangentPlane,
     projection_matrix);
    p_top = tangent_eye_pos(depths, tc_position + vec2(0.0, depths_size_inversed.y), tangentPlane,
     projection_matrix);
    p_bottom = tangent_eye_pos(depths, tc_position + vec2(0.0, -depths_size_inversed.y), tangentPlane,
     projection_matrix);
    vec3 dp_du = minimum_difference(ec_position, p_right, p_left);
    vec3 dp_dv = minimum_difference(ec_position, p_top, p_bottom) * (depths_size.y * depths_size_inversed.x);

```

```
    for (int i = 0; i < samples; ++i)
    {
#if USE_RANDOM_DIRECTION
        vec2 tc_sample_direction = random_rotation * poisson_disc[i];
#else
        vec2 tc_sample_direction = vec2(cos(float(i) * angle_step), sin(float(i) * angle_step));
#endif
        // Tangent vector
        vec3 ec_tangent = tc_sample_direction.x * dp_du + tc_sample_direction.y * dp_dv;
        float tan_tangent_angle = ec_tangent.z / length(ec_tangent.xy) + tan(bias);

        // Stepping
        vec2 tc_step_size = tc_sample_direction * tc_radius / float(steps);
        vec2 random_offset = tc_step_size * uniform_distribution_random;

        // Initialize horizon angle to the tangent angle
        float tan_horizon_angle = tan_tangent_angle;
        float sin_horizon_angle = tan_to_sin(tan_horizon_angle);

        for (float j = 0.0; j < float(steps); j += 1.0)
        {
            vec2 tc_sample = vec2(tc_position + tc_step_size * j + random_offset);
            vec3 ec_sample = get_ec_position(depths, tc_sample, projection_matrix);
            vec3 ec_ray = ec_sample - ec_position;
            float ec_ray_length_squared = dot(ec_ray, ec_ray);
            float tan_sample_angle = ec_ray.z / length(ec_ray.xy);

            bool in_hemisphere = ec_radius_squared >= ec_ray_length_squared;
            bool new_occluder = tan_sample_angle > tan_horizon_angle;

            if (in_hemisphere && new_occluder)
            {
                float sin_sample_angle = tan_to_sin(tan_sample_angle);
                float falloff = 1.0 - ec_ray_length_squared / ec_radius_squared;
                //float falloff = 1.0 - pow(min(sqrt(ec_ray_length_squared) / ec_radius, 1.0), 2.0);
                float horizon = sin_sample_angle - sin_horizon_angle;
                ambient_occlusion.a += horizon * falloff;
                tan_horizon_angle = tan_sample_angle;
                sin_horizon_angle = sin_sample_angle;

                bent_normal += normalize(ec_ray) * falloff;
            }
        }
    }

    bent_normal = normalize(bent_normal) * 0.5 + 0.5;
    ambient_occlusion.rgb = bent_normal;
    ambient_occlusion.a /= samples;
    ambient_occlusion.a = 1.0 - ambient_occlusion.a;
}
```

Listing 15: *HBAO*

Listing 16: *Direct Lighting and Ambient Occlusion*

```
#extension GL_NV_shader_buffer_load: enable
#extension GL_NV_gpu_shader5: enable
#extension GL_EXT_shader_image_load_store: enable

//#define USE_PHYSICAL_SOFT_SHADOWS
//#define USE_OREN_NAYAR_DIFFUSE_REFLECTANCE

const int poisson_disc_size = 16;
uniform vec2 poisson_disc[poisson_disc_size];

uniform sampler2D depths;
uniform sampler2D wc_normals, wc_positions;
```

```
13 uniform sampler2D albedos;
14 uniform sampler2D random;
15 uniform sampler2D ambient_occlusion;
16 uniform sampler2D shadow_map_0, shadow_map_1;
17 uniform sampler2D photon_splats, light_albedos;
18
19 uniform samplerBuffer lights;
20 #ifdef USE_TILED_SHADING
21 uniform isamplerBuffer light_grid;
22 uniform isamplerBuffer light_index_list;
23 uniform int tile_size;
24 #else
25 uniform int lights_size;
26 #endif
27
28 uniform ivec2 window_dimensions;
29 uniform ivec2 grid_dimensions;
30
31 uniform vec3 wc_view_eye_position;
32 uniform float z_near, z_far;
33
34 uniform mat4 projection_matrix;
35
36
37 struct view_type {
38     mat4 view_matrix, projection_matrix, view_projection_matrix;
39     vec4 eye;
40     vec4 right, forward, up;
41     ivec2 dimensions;
42 };
43
44 layout(std140) uniform user_view_block
45 { view_type user_view; };
46
47 uniform int ldm_view_count;
48
49 readonly restrict layout(std430) buffer view_block
50 { view_type views[]; };
51
52 readonly restrict layout(std430) buffer data_offset_block
53 { uvec4 data_offsets[]; };
54
55
56
57 struct light_type {
58     mat4 projection_matrix, view_projection_matrix;
59     vec4 wc_direction;
60     float radius;
61 };
62
63 layout(std140) uniform light_block
64 { light_type light; };
65
66
67
68 struct ldm_data {
69     uint32_t next;
70     float depth;
71 };
72 readonly restrict layout (std430) buffer data_buffer
73 { ldm_data data[]; };
74 readonly restrict layout (std430) buffer debug_view_buffer
75 { uint32_t debug_view[]; };
76 struct color_data {
77     uint32_t r, g, b;
78 };
```

```
readonly restrict layout (std430) buffer photon_splat_buffer
{ color_data photon_splats_data[]; };




struct vertex_data
{
    vec3 wc_view_ray_direction;
};
noperspective in vertex_data vertex;

layout(location = 0) out vec4 color;
//out vec4 overbright;



////////////////////////////////////////////////////////////////////////////////
/// Utility Functions
////////////////////////////////////////////////////////////////////////////////
float get_tc_z(
    in sampler2D sampler,
    in vec2 tc_position )
{ return texture(sampler, tc_position).z; }

float get_tc_z(
    in sampler2D sampler,
    in vec2 tc_position,
    in vec2 tc_offset )
{ return get_tc_z(sampler, tc_position + tc_offset); }

float get_ec_z(
    in float tc_z,
    in mat4 projection_matrix )
{ return projection_matrix[3][2] / (-2.0 * tc_z + 1.0 - projection_matrix[2][2]); }

float get_ec_z(
    in sampler2D sampler,
    in vec2 tc_position,
    in mat4 projection_matrix)
{ return get_ec_z(get_tc_z(sampler, tc_position), projection_matrix); }



////////////////////////////////////////////////////////////////////////////////
/// Shading functions
////////////////////////////////////////////////////////////////////////////////
float calculate_shadow_coefficient(
    in sampler2D shadow_map,
    in light_type light,
    in vec3 wc_position,
    in vec3 wc_normal,
    in vec2 tc_window,
    in float uniform_distribution_random )
{
    // Override
    light.radius = 5.0;

    // Normal bias (virtually translate scene along normal vectors)
    float cos_alpha = clamp(dot(wc_normal, light.wc_direction.xyz), 0.0, 1.0);
    float normal_bias_coefficient = sqrt(1.0 - cos_alpha * cos_alpha); // <=> sin(acos(wc_normal,
     light_direction));
    wc_position += wc_normal * normal_bias_coefficient * 1.5;

    // Coordinate transformations
    vec4 cc_position = light.view_projection_matrix * vec4(wc_position, 1.0);
```

```
    vec3 ndc_position = cc_position.xyz / cc_position.w;
    vec3 tc_position = (ndc_position + vec3(1.0)) * 0.5;
    float ec_position_z = get_ec_z(tc_position.z, light.projection_matrix);

    // Shadow behind light
    if (ec_position_z > 0.0) return 0.0;

    // Sample count as a function of light size
    int chi = int(light.radius / 12.5);
    int samples = clamp(chi * chi, 4, poisson_disc_size);

    // Random 2D rotation
    float alpha = uniform_distribution_random * PI * 2.0;
    mat2 random_rotation = mat2(cos(alpha), sin(alpha), -sin(alpha), cos(alpha));

    // Occluder search radius
    vec2 inverted_shadow_map_size = 1.0 / vec2(textureSize(shadow_map, 0));
    float tc_occluder_search_radius = light.radius * inverted_shadow_map_size;

#ifdef USE_PHYSICAL_SOFT_SHADOWS
    // Occluder search
    int occluder_count = 0;
    float ec_occluder_z = 0.0;
    for (int i = 0; i < samples; ++i)
    {
        vec2 tc_offset = random_rotation * poisson_disc[i] * tc_occluder_search_radius;

        float tc_occluder_sample_z = get_tc_z(shadow_map, tc_position.xy, tc_offset);
        float ec_occluder_sample_z = get_ec_z(tc_occluder_sample_z, light.projection_matrix);

        float ec_occluder_distance = ec_occluder_sample_z - ec_position_z;
        if (0.0 < ec_occluder_distance)
        {
            ec_occluder_z += ec_occluder_sample_z;
            ++occluder_count;
        }
    }
    // Return if no occluders were found
    if (0 == occluder_count) return 1.0;
    // Average occluder position
    ec_occluder_z /= occluder_count;
    // Distance from occluder to shading position
    float ec_occluder_distance = ec_occluder_z - ec_position_z;

    // Penumbra ratio relative to the light size (Calculated using similar triangles)
    float penumbra_ratio = ec_occluder_distance / ec_occluder_z;
#else
    const float penumbra_ratio = 0.05;
#endif
    // Sample for shadows in the penumbra
    float tc_shadow_sampling_radius = tc_occluder_search_radius * penumbra_ratio;

    // Shadow sampling (Using percentage-closer filtering)
    float shadow_coefficient = 0.0;
    for (int i = 0; i < samples; ++i)
    {
        vec2 tc_offset = random_rotation * poisson_disc[i] * tc_shadow_sampling_radius;
        float tc_occluder_sample_z = get_tc_z(shadow_map, tc_position.xy, tc_offset);

        if (tc_occluder_sample_z > tc_position.z)
            shadow_coefficient += 1.0;
    }
    return shadow_coefficient / float(samples);
}
```

```
210
211 vec3 fresnel_schlick( in vec3 specular_color, in vec3 wc_direction, in vec3 wc_half_angle )
212 {
213     return specular_color + (vec3(1.0) - specular_color)
214         * pow(1.0 - max(dot(wc_direction, wc_half_angle), 0.0), 5.0);
215 }
216
217
218
219 vec4 get_reflected_light(
220     in vec3 wc_position,
221     in vec3 wc_normal,
222     in vec3 wc_view_direction,
223     in vec3 albedo,
224     in vec3 bent_normal,
225     in float roughness )
226 {
227     vec3 wc_reflection = reflect(wc_view_direction, wc_normal);
228     // View direction is more convenient to store negated
229     wc_view_direction = -wc_view_direction;
230     // Common terms in the BRDFs
231     float a = roughness * roughness;
232     float a_squared = a * a;
233     vec3 material_specular_color = vec3(1.0, 1.0, 1.0);
234
235     vec4 result = vec4(0.0, 0.0, 0.0, 1.0);
236     const int count = 1;
237
238     for (int l = 0; l < count; ++l)
239     {
240         int light_id = l;
241
242         #define LIGHT_STRUCT_SIZE 6
243
244         vec3 wc_light_position = vec3(texelFetch(lights, light_id * LIGHT_STRUCT_SIZE).x, texelFetch(lights,
     light_id * LIGHT_STRUCT_SIZE + 1).x, texelFetch(lights, light_id * LIGHT_STRUCT_SIZE + 2).x);
245         vec3 light_color = vec3(texelFetch(lights, light_id * LIGHT_STRUCT_SIZE + 3).x, texelFetch(lights,
     light_id * LIGHT_STRUCT_SIZE + 4).x, texelFetch(lights, light_id * LIGHT_STRUCT_SIZE + 5).x);
246
247         float light_radius = 1.0;
248         light_color *= 400000.0;
249
250
251         // Representative point approximation of spherical lights
252         // Reference: http://www.unrealengine.com/files/downloads/2013SiggraphPresentationsNotes.pdf
253         vec3 wc_light_direction_unnormalized = wc_light_position - wc_position;
254         vec3 wc_center_to_reflection = dot(wc_light_direction_unnormalized, wc_reflection)
255             * wc_reflection - wc_light_direction_unnormalized;
256         vec3 wc_representative_point = wc_light_position
257             + wc_light_direction_unnormalized + wc_center_to_reflection
258             * clamp(light_radius / length(wc_center_to_reflection), 0.0, 1.0);
259
260
261         // Common BRDF parameters
262         vec3 wc_light_direction = wc_light_position - wc_position;
263         float wc_light_distance = length(wc_light_direction);
264         wc_light_direction /= wc_light_distance;
265         // The half angle vector (h)
266         vec3 wc_half_angle = normalize(wc_light_direction + wc_view_direction);
267         // Common dot products
268         float dotNH = dot(wc_normal, wc_half_angle);
269         float dotNV = dot(wc_normal, wc_view_direction);
270         float dotNL = dot(wc_normal, wc_light_direction);
271
272         // Spot light
273
```

```
        // Carpet Light
        vec3 wc_light_target = vec3(300.0, 100.0, -220.0);
        // Sky Light
        //vec3 wc_light_target = vec3(300.0, 600.0, -220.0);


        vec3 wc_direction = normalize(wc_light_position - wc_light_target);
        float eta = acos(max(dot(wc_light_direction, wc_direction), 0.0));
        if (eta > PI / 8.0) return vec4(0.0);

        // Falloff
        float falloff = 1.0 / (wc_light_distance * wc_light_distance);

        // Representative point normalization
        float a_prime = clamp(a + light_radius / (3.0 * wc_light_distance), 0.0, 1.0);
        float a_ratio = a / a_prime;
        float specular_sphere_normalization = a_ratio * a_ratio;

        // Specularly reflected light
        // Reference: http://www.unrealengine.com/files/downloads/2013SiggraphPresentationsNotes.pdf
        /*
        float chi = PI * (dotNH * dotNH * (a_squared - 1.0) + 1.0);
        float D = a_squared / (chi * chi);
        float k = (roughness + 1.0) * (roughness + 1.0);
        float Gv = dotNV / (dotNV * (1.0 - k) + k);
        float Gl = dotNL / (dotNL * (1.0 - k) + k);
        float G = Gv * Gl;
        vec3 F = fresnel_schlick(material_specular_color, wc_view_direction, wc_half_angle);
        vec3 specularly_reflected_light = (D * F * G) / (4.0 * dotNL * dotNV)
            * specular_sphere_normalization;
        */
        vec3 specularly_reflected_light = vec3(0.0);

        // Diffusely reflected light
        vec3 diffusely_reflected_light = albedo * max(dotNL, 0.0)
            * (vec3(1.0) - specularly_reflected_light);

#ifdef USE_OREN_NAYAR_DIFFUSE_REFLECTANCE
        // Reference: http://content.gpwiki.org/index.php/D3DBook:(Lighting)_Oren-Nayar
        float acos_dotNV = acos(dotNV);
        float acos_dotNL = acos(dotNL);
        float alpha = max(acos_dotNV, acos_dotNL);
        float beta = min(acos_dotNV, acos_dotNL);
        float gamma = dot(wc_view_direction - wc_normal * dotNV,
            wc_light_direction - wc_normal * dotNL);

        float A = 1.0 - 0.5 * a / (a + 0.33);
        float B = 0.45 * a / (a + 0.09);

        diffusely_reflected_light *= (A + (B * max(0.0, gamma) * sin(alpha) * tan(beta)));
#endif

        // Total reflected light
        result.rgb += (specularly_reflected_light + diffusely_reflected_light)
            * light_color * falloff;
    }

    return result;
}

vec4 get_view_color( in int view_id ) {
    const vec4 colors[3] = {vec4(0.0, 1.0, 0.0, 0.0), vec4(0.0, 0.0, 1.0, 0.0), vec4(1.0, 0.0, 0.0, 0.0)};
    return colors[view_id % 3];
}
```

```
vec4 get_debug_view() {
    uint32_t index = uint32_t(gl_FragCoord.x) + uint32_t(gl_FragCoord.y) * window_dimensions.x;

    if (0 < debug_view[index])
        return vec4(1.0);//get_view_color(int(debug_view[index] - 1));
    return vec4(0.0);
}



const float max_distance = 9999999.0;
float visibility( in float occluder_distance )
{ return (occluder_distance == max_distance) ? 1.0 : 0.0; }

const float falloff_distance = 200.0;
const float falloff_exponent = 2.0;
float attenuated_visibility( in float occluder_distance )
{ return pow(min(occluder_distance / falloff_distance, 1.0), falloff_exponent); }

float trace_ambient_occlusion( in int view_id, in vec3 wc_position, in vec3 wc_normal ) {
    view_type view = views[view_id];
    uint32_t data_offset = data_offsets[view_id];

    // Normal offset (virtually translate scene along normal vector)
    float cos_alpha = clamp(dot(wc_normal, view.forward.xyz), 0.0, 1.0);
    float normal_offset = sqrt(1.0 - cos_alpha * cos_alpha); // <=> sin(acos(cos_alpha));
    const float constant_factor = 10.0;
    wc_position += wc_normal * normal_offset * constant_factor;

    // Coordinate transformations
    vec4 cc_position = view.view_projection_matrix * vec4(wc_position, 1.0);
    if (cc_position.x > cc_position.w || cc_position.x < -cc_position.w
        || cc_position.y > cc_position.w || cc_position.y < -cc_position.w
        || cc_position.z > cc_position.w || cc_position.z < -cc_position.w)
        // Assume clear outside of LDM bounds
        return 1.0;

    vec3 ndc_position = cc_position.xyz / cc_position.w;
    vec3 tc_position = (ndc_position + vec3(1.0)) * 0.5;
    uvec2 sc_position = uvec2(tc_position.xy * view.dimensions);

    vec3 wc_eye = view.eye.xyz;
    vec3 right = view.right.xyz;
    vec3 forward = view.forward.xyz;
    vec3 up = view.up.xyz;

    // Othographic
    const float right_scale = 2000.0;
    const float top_scale = 2000.0;

    // Get the head node
    uint32_t head_index = data_offset + sc_position.x + sc_position.y * view.dimensions.x;
    uint32_t current = data[head_index].next;

    const int max_list_length = 200;
    float min_distance = max_distance;
    float previous_distance = min_distance;
    float next_distance = min_distance;
    bool get_next = false;

    int list_length = 0;
    while (0 != current && list_length++ < max_list_length) {
        float depth = data[current].depth;
        vec3 direction = (
            forward * (depth)
            + right * right_scale * ndc_position.x
```

```
            + up * top_scale * ndc_position.y);
        vec3 wc_sample_position = wc_eye + direction;

        float sample_distance = distance(wc_sample_position, wc_position);

        if (get_next) {
            get_next = false;
            next_distance = sample_distance;
        }

        if (sample_distance < min_distance) {
            previous_distance = min_distance;
            min_distance = sample_distance;
            get_next = true;
        } else break;

        current = data[current].next;
    }
    if (get_next) next_distance = max_distance;

    float cos_theta = dot(view.forward.xyz, wc_normal);

    return (cos_theta > 0.0)
        ? visibility(next_distance) * cos_theta
        : visibility(previous_distance) * -cos_theta;
}

float trace_ambient_occlusion( in vec3 wc_position, in vec3 wc_normal ) {
    float result = 0.0;
    for (int i = 0; i < ldm_view_count; ++i)
        result += trace_ambient_occlusion(i, wc_position, wc_normal);
    return 2.0 * result / float(ldm_view_count);
}


//#define DIRECT_LIGHT

void main()
{
    vec2 tc_window = gl_FragCoord.xy / window_dimensions;
    float ec_position_z = get_ec_z(depths, tc_window, projection_matrix);
    vec3 wc_position = wc_view_eye_position + vertex.wc_view_ray_direction * -ec_position_z / z_far;
    vec3 wc_normal = texture(wc_normals, tc_window).xyz;
    vec3 wc_view_direction = normalize(vertex.wc_view_ray_direction);
    vec3 albedo = texture(albedos, tc_window).xyz;

    float ambient_occlusion_factor = texture(ambient_occlusion, tc_window).a;

#ifdef DIRECT_LIGHT
    float roughness = 1.0;
    float uniform_distribution_random = texture(random, tc_window).x;

    vec3 bent_normal = normalize(texture(ambient_occlusion, tc_window).rgb * 2.0 - 1.0);
    const float a = 0.0;
    const float b = 1.0;
    const float c = 1.0;
    //ambient_occlusion_factor = pow(b * (ambient_occlusion_factor + a), c);

    // Direct light
    color = get_reflected_light(
        wc_position,
        wc_normal,
        wc_view_direction,
        albedo,
        bent_normal,
        roughness);
```

```

    // Shadow Mapping
    color.rgb *=
         calculate_shadow_coefficient(
            shadow_map_0,
            light,
            wc_position,
            wc_normal,
            tc_window,
            uniform_distribution_random);
#endif

    // Indirect light
    //color.rgb += 0.08 * albedo;
    //color.rgb += 0.08 * albedo * ambient_occlusion_factor;

    //vec4 environment_color = 0.05 * vec4(0.7, 0.7, 1.0, 1.0) * vec4(trace_ambient_occlusion(wc_position,
     wc_normal));
    //color += texture(photon_splats, tc_window) + environment_color;

    // Overrides
    //color.rgb = vec3(ambient_occlusion_factor);
    color = vec4(trace_ambient_occlusion(wc_position, wc_normal));
    //color += texture(photon_splats, tc_window);

    //color.rgb = texture(wc_positions, tc_window).xyz;
    //color = ldm();
    //color.rgb *= 0.1;
    //color =  1.0 * vec4(albedo, 0.0) + get_debug_view();

    //color = texture(light_albedos, tc_window);
    //color += sampling_test(wc_position);
    //color.rgb = wc_position;
    //color.rgb = albedo;
    //color.rgb = vec3(float(counts[index]) / 20.0);

    // Overbright
    const float bloom_limit = 1.0;
    vec3 bright_color = max(color.rgb - vec3(bloom_limit), vec3(0.0));
    float brightness = dot(bright_color, vec3(1.0));
    brightness = smoothstep(0.0, 0.5, brightness);
    //overbright.rgb = mix(vec3(0.0), color.rgb, brightness);
}
```

Listing 16: *Direct Lighting and Ambient Occlusion*

Listing 17: *Photon Tracing*

```
#extension GL_NV_gpu_shader5: enable

uniform sampler2D depths, wc_positions, wc_normals, light_depths, light_wc_normals, light_albedos;

uniform ivec2 window_dimensions;
uniform mat4 projection_matrix;
uniform float z_far;
uniform vec3 wc_view_eye_position;



struct view_type {
    mat4 view_matrix, projection_matrix, view_projection_matrix;
    vec4 eye;
    vec4 right, forward, up;
    ivec2 dimensions;
};

layout(std140) uniform current_view_block
```

```
20 { view_type current_view; };
21
22 layout(std140) uniform user_view_block
23 { view_type user_view; };
24
25 uniform int ldm_view_count;
26
27 readonly restrict layout(std430) buffer view_block
28 { view_type views[]; };
29
30 readonly restrict layout(std430) buffer data_offset_block
31 { uvec4 data_offsets[]; };
32
33
34
35 struct ldm_data {
36     uint32_t next;
37     float depth;
38 };
39 readonly restrict layout(std430) buffer data_buffer
40 { ldm_data data[]; };
41
42
43
44 layout(binding = 1, offset = 0) uniform atomic_uint photon_count;
45
46 struct photon_data {
47     vec4 wc_position, wc_normal, Du_x, Dv_x, radiant_flux;
48 };
49 coherent restrict layout(std430) buffer photon_buffer
50 { photon_data photons[]; };
51
52
53
54
55 struct vertex_data
56 {
57     vec3 wc_view_ray_direction;
58 };
59 noperspective in vertex_data vertex;
60
61
62
63 uint32_t compress( in vec4 clr )
64 { return (uint32_t(clr.x*255.0) << 24u) + (uint32_t(clr.y*255.0) << 16u) + (uint32_t(clr.z*255.0) << 8u) + (
     uint32_t(0.1*255.0)); }
65
66 vec4 decompress(uint32_t rgba)
67 { return vec4( float((rgba>>24u)&255u),float((rgba>>16u)&255u),float((rgba>>8u)&255u),float(rgba&255u) ) /
     255.0; }
68
69 float get_tc_z(
70     in sampler2D sampler,
71     in vec2 tc_position )
72 { return texture(sampler, tc_position).z; }
73
74 float get_ec_z(
75     in float tc_z,
76     in mat4 projection_matrix )
77 { return projection_matrix[3][2] / (-2.0 * tc_z + 1.0 - projection_matrix[2][2]); }
78
79 float get_ec_z(
80     in sampler2D sampler,
81     in vec2 tc_position,
82     in mat4 projection_matrix)
83 { return get_ec_z(get_tc_z(sampler, tc_position), projection_matrix); }
```

```
bool get_user_view_coordinates( in vec3 wc_position, out float ec_position_z, out ivec2 pc_position ) {
    vec4 cc_position = user_view.view_projection_matrix * vec4(wc_position, 1.0);
    if (cc_position.x > cc_position.w || cc_position.x < -cc_position.w
        || cc_position.y > cc_position.w || cc_position.y < -cc_position.w
        || cc_position.z > cc_position.w || cc_position.z < -cc_position.w)
        return false;
    vec3 ndc_position = cc_position.xyz / cc_position.w;
    vec2 tc_position = (ndc_position.xy + vec2(1.0)) * 0.5;
    ec_position_z = get_ec_z(depths, tc_position, user_view.projection_matrix);
    pc_position = ivec2(tc_position * user_view.dimensions);
    return true;
}


vec2 get_one_over_tan_half_fov( in mat4 projection_matrix )
{ return vec2(projection_matrix[0][0], projection_matrix[1][1]); }

vec2 get_tc_length( in float ec_length, in float ec_position_z, in mat4 projection_matrix )
{
    vec2 one_over_tan_half_fov = get_one_over_tan_half_fov(projection_matrix);
    return 0.5 * ec_length * one_over_tan_half_fov / -ec_position_z;
}

float K( float x ) {
    if (x >= 1.0) return 0.0;
    return 3.0 / PI * (1.0 - x * x) * (1.0 - x * x);
}

struct photon_differential
{ vec3 Du_x, Dv_x, Du_d, Dv_d; };

photon_differential construct_photon_differential( in vec3 d_hat, in vec3 right, in vec3 up ) {
    vec3 Du_d = (dot(d_hat, d_hat) * right - dot(d_hat, right) * d_hat) / pow(dot(d_hat, d_hat), 3.0 / 2.0);
    vec3 Dv_d = (dot(d_hat, d_hat) * up    - dot(d_hat, up   ) * d_hat) / pow(dot(d_hat, d_hat), 3.0 / 2.0);
    return photon_differential(vec3(0.0), vec3(0.0), Du_d, Dv_d);
}

void transfer( inout photon_differential photon, in vec3 d, in vec3 n, in float t ) {
    float Du_t = -dot((photon.Du_x + t * photon.Du_d), n) / dot(d, n);
    float Dv_t = -dot((photon.Dv_x + t * photon.Dv_d), n) / dot(d, n);

    photon.Du_x = (photon.Du_x + t * photon.Du_d) + Du_t * d;
    photon.Dv_x = (photon.Dv_x + t * photon.Dv_d) + Dv_t * d;
}

vec4 alpha( in vec3 w_i, in vec3 w_o )
{ return normalize(vec4(cross(w_i, w_o), 1.0 + dot(w_i, w_o))); }

// See https://code.google.com/p/kri/wiki/Quaternions for reference
vec3 rotate_vector( vec4 q, vec3 v )
{ return v + 2.0 * cross(q.xyz, cross(q.xyz, v) + q.w * v); }

void diffusely_reflect( inout photon_differential photon, in vec3 w_i, in vec3 w_o ) {
    vec4 q = alpha(w_i, w_o);
    photon.Du_d = rotate_vector(q, photon.Du_d);
    photon.Dv_d = rotate_vector(q, photon.Dv_d);
}


void store_photon( in vec3 wc_position, in vec3 wc_normal, in photon_differential photon, in vec4 radiant_flux
    ) {
    vec3 abs_x = max(abs(photon.Du_x), abs(photon.Dv_x));
    float max_x = max(max(abs_x.x, abs_x.y), abs_x.z);
```

```
    const float photon_footprint_bias = 0.5;
    if (photon_footprint_bias < max_x) return;

    uint32_t id = atomicCounterIncrement(photon_count);

    photons[id].wc_position = vec4(wc_position, 1.0);
    photons[id].wc_normal = vec4(wc_normal, 0.0);
    photons[id].Du_x = vec4(photon.Du_x, 0.0);
    photons[id].Dv_x = vec4(photon.Dv_x, 0.0);
    photons[id].radiant_flux = vec4(radiant_flux.rgb, 1.0);
}


void store_first_bounce( in photon_differential photon, in vec3 wc_position, in vec3 wc_x, in vec3 wc_normal,
     in vec3 w_i, in vec3 w_o, in vec4 radiant_flux_and_f ) {
    const float const_bias = 0.1;
    float ec_z_actual = (user_view.view_matrix * vec4(wc_x, 1.0)).z;

    // Project into user view
    float ec_z_seen_by_user;
    ivec2 pc_user_position;
    if (get_user_view_coordinates(wc_x, ec_z_seen_by_user, pc_user_position)
        && ec_z_seen_by_user < ec_z_actual + const_bias)
    {
        vec3 wc_hit_normal = texture(wc_normals, vec2(pc_user_position) / vec2(user_view.dimensions)).xyz;
        float t = distance(wc_position, wc_x);

        diffusely_reflect(photon, w_i, w_o);
        transfer(photon, w_o, wc_hit_normal, t);

        float cos_theta = dot(wc_normal, w_o); // [sr]
        if (0.0 < cos_theta)
            store_photon(wc_x, wc_hit_normal, photon, radiant_flux_and_f * cos_theta);
    }
}

void store_first_bounce_in_both_directions( in int view_id, in photon_differential photon, in vec3 wc_position,
      in vec3 wc_normal, in vec4 radiant_flux_and_f ) {
    view_type view = views[view_id];
    uint32_t data_offset = data_offsets[view_id];

    // Coordinate transformations
    vec4 cc_position = view.view_projection_matrix * vec4(wc_position, 1.0);
    if (cc_position.x > cc_position.w || cc_position.x < -cc_position.w
        || cc_position.y > cc_position.w || cc_position.y < -cc_position.w
        || cc_position.z > cc_position.w || cc_position.z < -cc_position.w)
        return;
    vec3 ndc_position = cc_position.xyz / cc_position.w;
    vec3 tc_position = (ndc_position + vec3(1.0)) * 0.5;
    uvec2 pc_position = uvec2(tc_position.xy * view.dimensions);

    vec3 wc_eye = view.eye.xyz;
    vec3 right = view.right.xyz;
    vec3 forward = view.forward.xyz;
    vec3 up = view.up.xyz;

    // Othographic
    const float right_scale = 2000.0;
    const float top_scale = 2000.0;

    // Get the head node
    uint32_t heads_index = data_offset + pc_position.x + pc_position.y * view.dimensions.x;
    uint32_t current = data[heads_index].next;

    const int max_list_length = 2048;
    const float FLOAT_MAX = 999999.0;
```

```
    float min_distance = FLOAT_MAX;
    float previous_distance = min_distance;
    float next_distance = min_distance;
    vec3 wc_sample_position,
        wc_last_sample_position,
        wc_previous,
        wc_next;
        uint32_t sample_diffuse, last_diffuse, previous_diffuse, next_diffuse;
    bool get_next = false;

    int list_length = 0;
    while (0 != current && list_length < max_list_length) {
        float depth = data[current].depth;

        vec3 direction = (
            forward * depth
            + right * right_scale * ndc_position.x
            + up * top_scale * ndc_position.y);
        wc_sample_position = wc_eye + direction;

        float sample_distance = distance(wc_sample_position, wc_position);

        if (get_next) {
            get_next = false;
            next_distance = sample_distance;
            wc_next = wc_sample_position;
        }

        if (sample_distance < min_distance) {
            previous_distance = min_distance;
            wc_previous = wc_last_sample_position;

            min_distance = sample_distance;
            get_next = true;
        } else break;

        wc_last_sample_position = wc_sample_position;

        current = data[current].next;
        ++list_length;
    }
    if (get_next) wc_next = vec3(FLOAT_MAX);

    if (1 >= list_length) return;


    vec3 w_i = normalize(-vertex.wc_view_ray_direction);

    store_first_bounce(photon, wc_position, wc_previous, wc_normal, w_i, normalize(-forward),
     radiant_flux_and_f);
    store_first_bounce(photon, wc_position, wc_next,     wc_normal, w_i, normalize(forward),
     radiant_flux_and_f);
}


void main()
{
    // Convert to spot light
    float radius = length(gl_FragCoord.xy / window_dimensions - vec2(0.5));
    if (radius > 0.5) discard;

    // Deferred parameters
    const ivec2 light_view_dimensions = ivec2(100);
    const vec2 window_scale_constant = vec2(light_view_dimensions) / float(user_view.dimensions);
    vec2 tc_window = gl_FragCoord.xy / window_dimensions * window_scale_constant;
    float ec_position_z = get_ec_z(light_depths, tc_window, projection_matrix);
```

```
    vec3 wc_position = wc_view_eye_position + vertex.wc_view_ray_direction * -ec_position_z / z_far;
    vec4 rho_d = texture(light_albedos, tc_window); // [1]
    vec3 wc_normal = texture(light_wc_normals, tc_window).xyz;
    const vec4 light_radiant_flux = vec4(vec3(1000000000.0), 1.0); // [W]
    const int light_view_size = light_view_dimensions.x * light_view_dimensions.y;
    const float spot_light_ratio = PI / 4.0; // A_circle / A_square
    const int photon_count = int(light_view_size * 2 * ldm_view_count * spot_light_ratio);
    const vec4 radiant_flux = light_radiant_flux / float(photon_count); // [W]

    // Rename for consistency with theory
    vec3 x = wc_view_eye_position;
    vec3 n = wc_normal;
    vec3 d_hat = vertex.wc_view_ray_direction;
    vec3 d = normalize(d_hat);
    vec3 right = current_view.right.xyz;
    vec3 up = current_view.up.xyz;

    // Ray differential construction and initial transfer
    photon_differential photon = construct_photon_differential(d_hat, right, up);

    float t = -ec_position_z;//-dot(x, n) / dot(d, n);
    transfer(photon, d, n, t);

    // Direct photon
    //store_photon(wc_position, wc_normal, photon, radiant_flux);

    // 1. bounce
    vec4 f = rho_d / PI; // [sr^-1]
    for (int i = 0; i < ldm_view_count; i++)
        store_first_bounce_in_both_directions(i, photon, wc_position, wc_normal, radiant_flux * f);
}
```

Listing 17: *Photon Tracing*

Listing 18: *Photon Splatting Vertex Shader*

```
uniform mat4 view_matrix;
uniform mat4 view_projection_matrix;
uniform uint photon_count;



layout(location = 0) in vec3 wc_position;
layout(location = 1) in vec3 wc_normal;
layout(location = 2) in vec3 Du_x;
layout(location = 3) in vec3 Dv_x;
layout(location = 4) in vec4 radiant_flux;

out photon_data {
    vec3 wc_position, Du_x, Dv_x;
    vec4 irradiance;
    mat3 M;
} photon;

mat3 mat3_from_rows( in vec3 row1, in vec3 row2, in vec3 row3 ) {
    return mat3(
        row1.x, row2.x, row3.x,  // Column 1
        row1.y, row2.y, row3.y,  // Column 2
        row1.z, row2.z, row3.z); // Column 3
}

void main() {
    gl_Position = view_projection_matrix * vec4(wc_position, 1.0);
    photon.wc_position = wc_position;

    const float scale = 2000.0;
    photon.Du_x = Du_x * scale;
```

```
    photon.Dv_x = Dv_x * scale;

    const float a = scale;
    photon.M = mat3_from_rows(
        cross(photon.Dv_x, wc_normal),
        cross(wc_normal, photon.Du_x),
        a * wc_normal);
    photon.M *= 2.0 / dot(photon.Du_x, cross(photon.Dv_x, wc_normal));

    float area = PI / 4.0 * length(cross(photon.Du_x, photon.Dv_x)); // [m^2]
    photon.irradiance = radiant_flux / area; // [W * m^-2]
}
```

Listing 18: *Photon Splatting Vertex Shader*

Listing 19: *Photon Splatting Geometry Shader*

```
uniform mat4 view_projection_matrix;



layout(points) in;
layout(triangle_strip, max_vertices = 4) out;

in photon_data {
    vec3 wc_position, Du_x, Dv_x;
    vec4 irradiance;
    mat3 M;
} photon[];

out splat_data {
    flat vec3 wc_position;
    flat mat3 M;
    flat vec4 irradiance;
} splat;


vec4 cc_position( in vec3 wc_offset )
{ return view_projection_matrix * vec4(photon[0].wc_position + wc_offset, 1.0); }

void main() {
    splat.wc_position = photon[0].wc_position;
    splat.irradiance = photon[0].irradiance;
    splat.M = photon[0].M;

    gl_Position = cc_position(0.5 * (-photon[0].Du_x - photon[0].Dv_x));
    EmitVertex();

    gl_Position = cc_position(0.5 * (-photon[0].Du_x + photon[0].Dv_x));
    EmitVertex();

    gl_Position = cc_position(0.5 * ( photon[0].Du_x - photon[0].Dv_x));
    EmitVertex();

    gl_Position = cc_position(0.5 * ( photon[0].Du_x + photon[0].Dv_x));
    EmitVertex();
}
```

Listing 19: *Photon Splatting Geometry Shader*

Listing 20: *Photon Splatting Fragment Shader*

```
uniform sampler2D wc_positions, albedos;
uniform float specular_exponent;
uniform ivec2 window_dimensions;


in splat_data {
```

```
    flat vec3 wc_position;
    flat mat3 M;
    flat vec4 irradiance;
} splat;

layout(location = 0) out vec4 radiance;



float K( float x ) {
    return (1.0 > x)
        ? 3.0 / PI * (1.0 - x * x) * (1.0 - x * x)
        : 0.0;
}

void main() {
    vec2 tc_position = gl_FragCoord.xy / vec2(window_dimensions);
    vec3 wc_position = texture(wc_positions, tc_position).xyz;
    vec4 albedo = texture(albedos, tc_position); // [1]

    float d = length(splat.M * (wc_position - splat.wc_position));
    vec4 f = albedo / PI; // [sr^-1]

    radiance = PI * K(d) * f * splat.irradiance; // [W * m^-2 * sr^-1]
}
```

Listing 20: *Photon Splatting Fragment Shader*

This thesis was prepared at the department of Applied Mathematics and Computer Science (DTU Compute) at the Technical University of Denmark (DTU) in fulfilment of the requirements for acquiring an M.Sc. in Mathematical Modelling and Computing (MMC).

Technical University of Denmark
Department of Applied Mathematics and Computer Science

Richard Petersens Plads, Building 324,
2800 Kongens Lyngby, Denmark

Phone +45 4525 3031
CVR 30 06 09 46
EAN 5798000428515

compute@compute.dtu.dk
www.compute.dtu.dk

Frederik Peter Aalund Lyngby, February 1, 2015

DTU Compute—M.Sc.—February, 2015