UTRECHT UNIVERSITY

MASTER THESIS

Fast Preprocessed Global Illumination for Massive Scenes with Live Updates

Author:
Casper SCHOULS
ICA-4017544

Supervisors:
dr. ing. Jacco BIKKER
dr. A. VAXMAN

September 28, 2017

Utrecht University

Utrecht University

Abstract

Fast Preprocessed Global Illumination for Massive Scenes with Live Updates

by Casper SCHOULS

This master thesis project was performed under supervision of dr. J. Bikker and dr. A. Vaxman at Utrecht University, in collaboration with Stirling Labs Ltd. First, we introduce our problem of precomputing global illumination on massive scenes consisting of over a billion triangles. A broad literature study evaluates several key aspects of dealing with this problem, such as implementing acceleration structures for ray tracing, constructing photon maps and evaluating the rendering equation.

We propose a global illumination pipeline that starts with computing global illumination using a photon map. The radiance estimate over the surface of the mesh is then stored in a texture, which is used to generate an indirect illumination map using final gather. The indirect illumination map is composited at runtime with direct illumination or combined with direct illumination and stored on disk.

Using live updates to our indirect illumination map, the initial lighting is improved at runtime. Our method is evaluated by testing our acceleration structures, testing our global illumination pipeline performance and comparing our lighting with reference images produced using a path tracer. Produced lighting is of sufficient quality for our purposes and the global illumination pipeline is performant enough for the current stage of development.

Furthermore, we use this opportunity to pursue research into the practical time complexity of ray traversal using a bounding volume hierarchy in a seperate small paper, included as an appendix. Our results show that assumed logarithmic time scaling for ray traversal is an underestimate of real-world ray traversal speed. Furthermore, we show that ray traversal speed starts to deteriorate more rapidly after the CPU caches are flooded.

Contents

Chapter 1

Introduction

Rendering massive virtual scenes, consisting of multiple billions of triangles, is computationally expensive. To do so for virtual-reality applications that use head-mounted displays, such as the Oculus Rift [26] and the HTC Vive [7] is even more complex, as these typically have high resolutions (HD+) and require high framerates (90+ FPS) to avoid motion sickness. To achieve a sense of realism, the massive scenes must be illuminated using physically-based light models.

This master project considers these massive scenes, created by architects for megastructures. These megastructures are to be previsualized using virtual-reality headsets, during the design process. In this manner, architects can preview the result of their work, before physical construction even begins. In cooperation with Stirling Labs Ltd [1], the goal of this master project is to precompute global illumination for rendering these megastructures. Our use case requires the precomputation process to be fast: designers of these megastructures will not want to wait hours before being able to visualize their changes. Eventually, our system should be able to precompute global illumination within an hour. Furthermore, lighting must be accurate as our system will be used to make decisions in the design process. However, trade-offs are possible: if some lighting quality can be sacrificed for large performance gains, this is considered.

1.1 Objectives

We have specifically chosen to precompute global illumination rather than work with real-time global illumination. The scenes we are using are massive and we therefore expect that modern real-time global illumination algorithms, such as image-space photon mapping by McGuire et al. [29] or voxel cone tracing by Crassin et al. [8] are not fast enough for our scenes. Instead, we precompute global illumination and use the precomputed lighting in a rasterizer or ray tracer.

Implementing such a global illumination system requires the use of acceleration structures for ray tracing. The opportunity is taken to pursue research into the time scaling of ray traversal through a *bounding volume hierarchy* (BVH). A seperate small paper is included with this thesis that presents our experiments and results. The results are used to predict how our global illumination system will perform on massive scenes. Furthermore, ray tracing is investigated as a solution for determining primary visibility of polygons in the scene.

1.2 Research Questions

The following research questions have been formulated with regards to the project introduced above.

1. How can global illumination for scenes consisting of billions of triangles be precomputed fast enough for prototyping purposes described above?
   (a) Is photon mapping a suitable method for fast precomputation of global illumination on massive scenes?
2. Does the theoretical logarithmic complexity of BVH ray traversal hold up for very large scenes?

1.3 High-level Approach

We have chosen to implement a combination of photon mapping [22] and light map baking. Photon mapping uses ray tracing to estimate the distribution of photons in the scene. Interpolating from this distribution of photons, an estimate of the global illumination at any surface point can be made. This *radiance estimate* is cached and used to determine indirect lighting using *final gather*. Direct lighting is computed seperately and added to the indirect lighting derived from the photon map. The resulting global illumination is baked in a light map and used in a forward renderer or ray tracing renderer. At runtime, our system can incrementally update the lighting, sacrificing initial lighting quality for faster start-up times.

1.4 Content Overview

We first start with an overview of preliminaries and related work in Chapter 2. In Chapter 3, our implementation, as described in Section 1.3, is presented in more detail. Relevant concepts from the literature are linked to our own implementation. Chapter 4 presents a thorough evaluation of our work: we measure performance of our global illumination pipeline as well as our ray tracing acceleration structures. Furthermore, the quality of the produced lighting is evaluated. Here, future work is also suggested for each component of our implementation. The thesis concludes by answering the research questions posed in Section 1.2 and discussing potential future directions for the project.

Chapter 2

Related work

In this chapter, related work is discussed that is relevant to rendering and global illumination. We start with a quick overview of what rendering entails before exploring ray tracing further. We show how implementing efficient ray tracing algorithms is relevant for our global illumination method. We proceed to discuss how ray tracing can be accelerated. After this, we discuss what global illumination is and consider algorithms that are capable of computing global illumination, either in real-time or precomputed.

2.1 Rendering

By rendering, we denote the process by which an image is generated from a digital description of a scene, given a camera position within this scene. The output of a rendering algorithm is a 2D image or *render* of the scene, as visible from the viewpoint of the camera.

A scene often consists of polygonal geometry which is grouped into models. Furthermore, light sources have to be defined which illuminate the scene. These models, together with the light sources and materials make up the description of the scene. In many applications, the polygonal geometry is defined as a set of triangles. Rendering algorithms typically determine which triangles are projected onto which pixels of the output image. After this projection is complete, the color of these pixels is determined using information about the geometry, lighting and materials in the scene. The process of determining the color of the pixel is referred to as *shading*.

An important aspect of shading is to determine how each rendered surface point is illuminated. This illumination can arrive directly from light sources, or indirectly by bouncing off other geometry in the scene. The total illumnation arriving at a surface point is referred to as *global illumination*. Global illumination and shading will be the topics of Section 2.3.

Traditionally, two rendering algorithms can be discerned, rasterization and ray tracing. The first is discussed briefly in Section 2.1.1, the latter is discussed in Section 2.1.2.

2.1.1 Rasterization

Rasterization is the most widely used algorithm for real-time rendering. Rasterization projects polygonal geometry towards the virtual camera by multiplying a *projection matrix* with the 3D vertices of the polygonal geometry. The result of this multiplication is the coordinate of the pixel that the 3D vertex ends up on. Rasterization thus projects 3D *world-space coordinates* to 2D *screen-space coordinates*. After this projection,

information about the original 3D position of the vertex can be combined with information about lighting in the scene to determine which color the output pixel should have. Rasterization is used in all modern graphics cards and in most modern games.

Rasterization is limited in taking into account global information about the scene. Each triangle and surface point is processed individually by matrix multiplication, in the process disregarding relevant information about other geometry. This makes it hard to model global effects such as reflections, refractions and indirect lighting which depend upon other geometry in the scene.

2.1.2 Ray Tracing

Ray tracing is a rendering algorithm that does not process each piece of polygonal geometry individually, but instead, casts rays to determine visibility of polygons. Each ray corresponds to a pixel on the 2D output image. These rays are checked for intersections with the polygonal geometry in the scene using ray/polygon intersection tests. If a ray intersects the polygon, and this is the nearest intersection of all intersected polygons, it follows that the polygon in question is visible on the pixel of the 2D output image. Because we now know which polygon is visible there, we can use information about the ray/polygon intersection point and the lighting in the scene to perform shading. If necessary, additional rays can be cast from the intersection point to gather global information. Because each polygon in the scene needs to be checked against each ray cast from the camera, ray tracing is a computationally expensive algorithm.

Our technique uses *photon mapping*, which tracks photons from the light sources and continues tracing their paths as they bounce through the scene. The individual path segments of the photons can be modelled as rays and therefore, ray tracing algorithms can be used to effectively determine the complete photon path. It is therefore crucial to develop a fast implementation of ray tracing.

Many extensions exist to both rasterization and ray tracing, which are aimed at increasing the rendering speed and producing higher visual quality in the output images. In this section, ray tracing is discussed extensively as it is required to implement an efficient ray tracing algorithm for both preprocessing lighting and rendering the scene. First, in Section 2.1.2, *Whitted-Style Ray Tracing* is discussed. Acceleration structures that aim to speed up ray tracing are discussed in Section 2.2.

Whitted-Style Ray Tracing

Whitted-Style ray tracing was introduced in a 1980 paper by Whitted [43]. It broke new ground as it used global information to compute shading.

Whitted's approach is to cast additional rays from the intersection points of the rays used for determining visibility. The rays that are used for determining visibility, spawned in the camera, are commonly referred to as *primary rays*. Additional rays that are cast are referred to as *extension rays* or *secondary rays*. By casting additional rays, we are able to accurately render reflections, refractions and shadows shown in Figure 2.1.

While Whitted-style ray tracing was extremely slow when it was first implemented in 1980, it is now possible to run this in real-time on modern programmable graphics hardware as is shown in for instance, Gunther et al. [16].

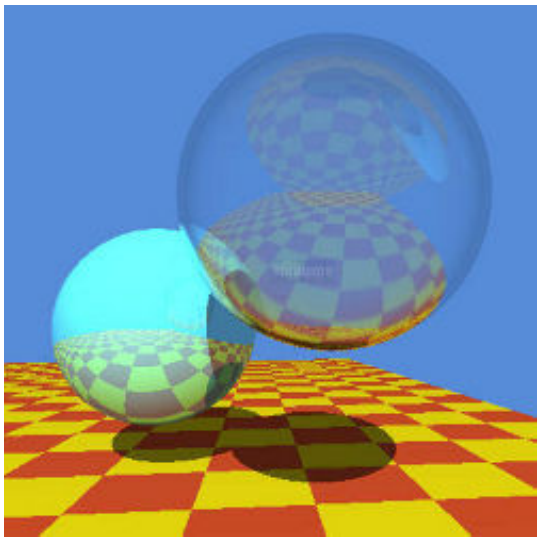

FIGURE 2.1: An image generated by T. Whitted for his 1980 paper[43], clearly showcasing shadow, reflection and refraction.

2.2 Acceleration Structures For Ray Tracing

In ray tracing, we need to check each ray that is cast against each polygon in the scene. This incurs massive computational cost, as both the number of rays and the number of polygons are typically very large. In order to speed this up, the polygons in the scene can be organized in an acceleration structure. Such acceleration structures allow us to quickly reject large groups of polygons of which we can be certain that the ray does not intersect them. This technique is referred to as *intersection culling*.

In general, acceleration structures either divide up the space in which the objects reside or they group the objects together in bounding volumes. In this manner, either large areas of the space can be culled quickly or large groups of objects can be culled early.

In this section, we review several often used acceleration structures for ray tracing. In Section 2.2.1, the *uniform grid* and *brickmap* will be discussed. These divide space up in cells, which contain the polygons. Section 2.2.2 considers the *KD-tree*, which subdivides space recursively using splitting planes. Then, Section 2.2.3 considers the *Bounding Volume Hierarchy* (BVH), which groups polygons together in a hierarchical tree.

Note that extensive comparisons of acceleration structures for ray tracing have already been performed. See, for example, Havran et al. [18] or Thrane et al. [34]. These focus mostly on the (uniform) grid and the KD-tree. Much recent research has focused on implementing the BVH and this has established the BVH as the state-of-the-art for modern ray tracing. For example, Overbeck. et al [30] propose a fast BVH-traversal algorithm and Wald et al. [39] propose a fast construction algorithm for the BVH.

2.2.1 Grid

One of the most basic acceleration structures that can be used for ray tracing is the uniform grid. The uniform grid subdivides space into equally-sized three-dimensional

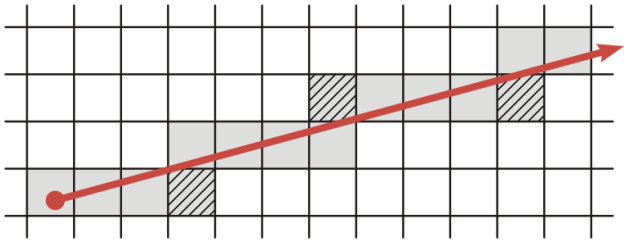

FIGURE 2.2: An illustration of the tested voxels during ray traversal.

cubes and is thus a spatial subdivision datastructure. These cubes are often referred to as *cells* or *voxels*.

Grid Construction

Each cell in a uniform grid contains a list of triangles which are contained fully or partially within the cell. Grid construction time scales linearly with the number of polygons in the scene, which makes it one of the fastest construction processes. The uniform grid is able to handle changes in geometry easily, by updating the list of triangle pointers in the affected voxels.

Grid Traversal

In 1987, Amanatides and Woo introduce a fast voxel traversal algorithm [2] that traverses uniform grids by determining the next voxel along the ray using the ray direction and the current voxel coordinates. In this manner, traversal can stop as soon as the first intersecting voxel is found, as geometry beyond the current voxel will be hidden from view by the current voxel.

One of the main drawbacks of the uniform grid is that the memory access patterns are incoherent. This incoherent memory access pattern can be aleviated by applying packet-based traversal to grid, as shown by Wald et al. in 2006 [42]. The uniform grid also suffers from the *teapot in a stadium* problem. This problems refers to a scene with high geometric complexity in the centre (near the teapot) and low geometric complexity in the large stadium surrounding the teapot. If a very fine grid resolution is chosen, a ray starting near the edge of the stadium must traverse many empty cells before eventually reaching the teapot, incurring computation overhead. When it does eventually reach the teapot, the teapot geometry can be checked for intersections quickly due to the fine grid resolution. However, when a very coarse grid resolution is chosen, the ray will reach the teapot quickly, but when it does, must spend much time testing all of the teapot geometry against the ray because of the coarse grid resolution.

Brickmap

A brickmap is a refinement of the uniform grid that subdivides individual cells of the uniform grid further. In this manner, areas with high geometric complexity can be refined with a more dense grid and areas with low geometric complexity can be left at coarse resolutions. Referring to the teapot in a stadium, this allows us to coarsely subdivide the space around the teapot, while finely subdividing the space that contains the teapot. A brickmap datastructure can be an effective alternative

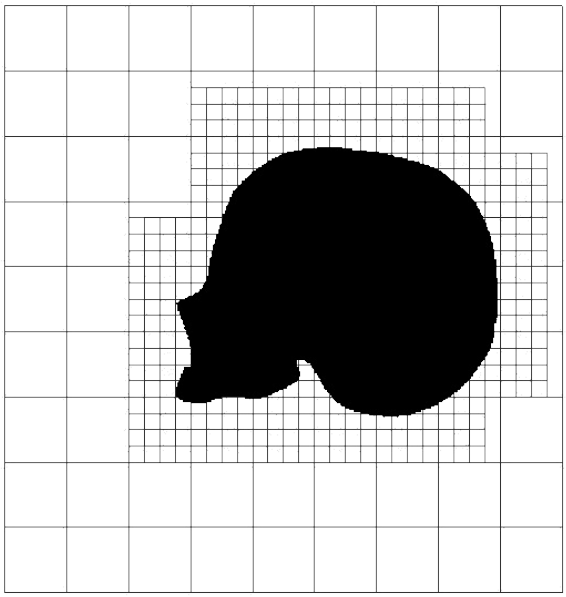

FIGURE 2.3: Space subdivision in a brickmap around a skull. Image taken from Fairlight's post. [12]

to a uniform grid or more complex acceleration structures such as the BVH or the KD-tree. This was demonstrated by *Fairlight's* real-time ray tracing demo *5 faces* [11, 12], which uses a brickmap to render animated scenes in real-time.

2.2.2 KD-tree

A *KD-tree* is a spatial subdivision structure that recursively subdivides space using splitting planes. KD-trees can be constructed automatically by recursively determining the best splitting plane. These are stored as a hierarchical set of nodes in a binary tree. Each voxel in the KD-tree contains a list of pointers to the polygons that it contains.

KD-tree Traversal

Basic KD-tree traversal [21] starts at the root of the KD-tree. For each interior node in the tree, it is determined which of the two child nodes is nearer and which is further along the ray. Both of these nodes are checked in turn for intersection with the enclosing voxels. If the ray intersects the child voxel, it is traversed as well. When a leaf node is reached, the polygons inside the voxel are processed individually.

KD-tree Construction

Automatic KD-tree construction starts by subdividing the root node using a splitting plane. The splitting plane is a plane that splits the set of polygons in the current voxel in two subsets. This splitting plane can be determined automatically using the *surface area heuristic* (SAH)[27]. For more on the SAH, see section 2.2.3. KD-tree building has been investigated further by Wald and Havran [38] and they have derived an algorithm that has many similarities to their BVH construction algorithm.

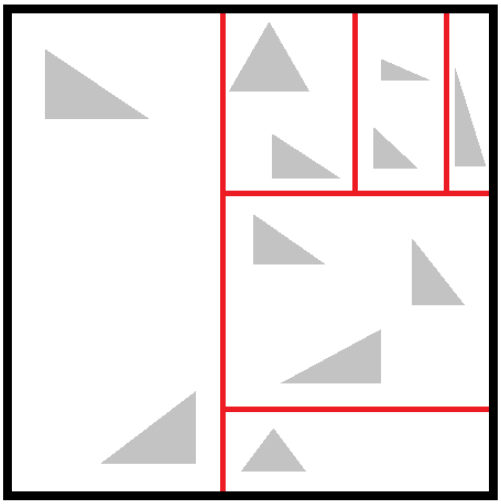

FIGURE 2.4: A possible spatial subdivision of a KD-tree. Polygons are shown in grey, splitting planes in red.

In Listing 2.2, a general algorithm for BVH building is shown. A similar algorithm can be applied for KD-tree building, using the same heuristics.

2.2.3 Bounding Volume Hierarchy

The *Bounding Volume Hierarcy*[32] is an object subdivision structure. Polygons are grouped and encapsulated within bounding volumes, typically axis-aligned bounding boxes. The idea here is that during ray traversal, if a ray does not intersect a bounding volume, all contained polygons are also not intersected and can be culled.

As the name implies, these groups of polygons are stored in a hierarchical tree. The tree is typically binary, although variations exist with higher branching factors. In the interior nodes, bounding volumes are stored along with pointers to the two child nodes. The leaf nodes contain a list of polygons, along with a bounding volume that fully encloses these polygons. In a tree with a higher branching factor, each interior node may have more than two children. Such a tree is referred to as a *Multi-BVH* or *MBVH*. This was first proposed by Dammertz et al. [9], with Wald et al. [40] proposing an optimized, SIMD-enabled traversal strategy for the MBVH.

BVH Traversal

During ray intersection testing, the BVH is traversed top down. First, the root node's bounding volume is tested for intersection with the ray. If it intersects, the two child node bounding volumes are tested. This process repeats as we move down the tree until the leaf nodes are reached, in which the containing polygons are tested for intersection with the ray. Pseudocode for basic, recursive BVH traversal is given below in Listing 2.1.

```
void TraverseBVH(Node node, Ray ray)
    if(node.isInner())
        if(ray.Intersects(node.BoundingVolume))
            TraverseBVH(node.LeftChild, ray);
            TraverseBVH(node.RightChild, ray);
```

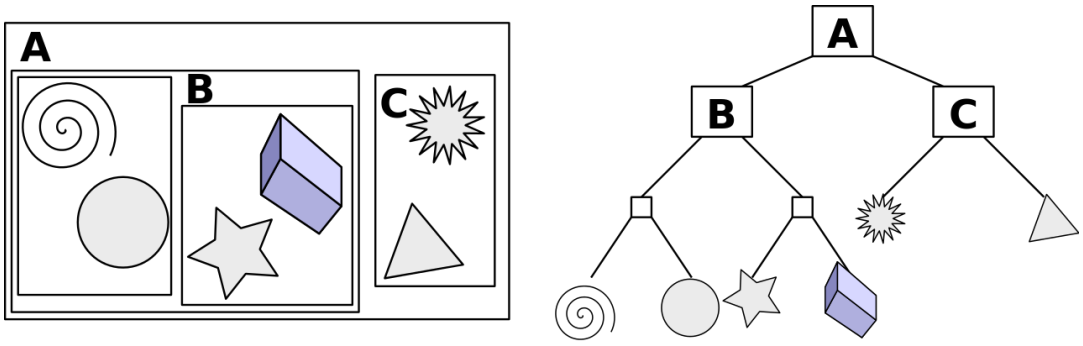

FIGURE 2.5: An illustration of a scene with bounding volumes (left) and the corresponding BVH (right). Each bounding volume is labeled with a letter and each polygon is represented by a primitive.

```
else //Process leaf polygons
    foreach(Polygon polygon in node.Polygons)
        if(ray.Intersects(polygon))
            return;
```

LISTING 2.1: Basic recursive BVH traversal

Basic BVH traversal traces one ray through the BVH at a time. While this is a simple and straightforward approach, it suffers from several problems. Firstly, memory access is incoherent like the grid. Secondly, the algorithm is recursive which generally performs less optimal than an iterative implementation. Lastly, basic BVH traversal does not take into account the order of the child nodes during traversal, which makes culling less efficient. Fortunately, the algorithm can trivially be converted to an iterative version using a stack. A visualization of BVH traversal steps can be seen in Figure 2.6, depicting a model of a slug. Green indicates that the ray had to traverse relatively few interior nodes before intersecting the polygon. Red indicates that the ray had to traverse many nodes.

This basic iterative algorithm lies at the basis of many traversal algorithms that have been proposed over the years. In 2001, I. Wald [41] proposed a vectorized approach to BVH traversal, which is especially suited for rays that take the same path through the BVH. In 2007, Overbeck et al. [30] published a new algorithm that traces large packets of rays through the BVH at once, while making use of SIMD registers. This has the added benefit of amortizing memory access cost over the ray packet and culling large packets of rays at once using a frustum culling test.

Much recent research has focused on improving performance for incoherent ray sets, such as shadow rays or extension rays, as these are common in path tracing (see Section 2.3.2). Examples are dynamic ray stream traversal [3] and ordered ray stream traversal [13]. A notable recent approach that claims significant performance improvements for primary ray traversal is *coherent large packet traversal* [13], which traverses a BVH with a branching factor of four instead of two using precomputed orderings of child nodes, stored in a lookup table. This method appears to be the *state of the art* for BVH traversal.

BVH Construction

When Rubin et al. published the original BVH paper in 1980 [32], they proposed manual construction of the BVH by selecting polygons for each bounding volume.

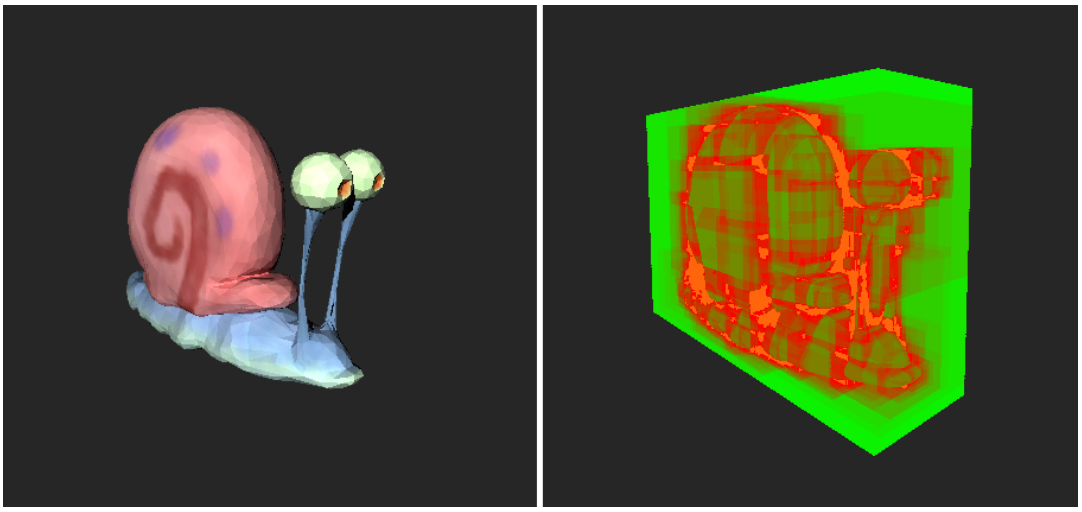

FIGURE 2.6: A ray traced image of a slug (left), compared to it's BVH traversal steps visualization (right). Green indicates few traversal steps, while red indicates many.

However, research has shown that high-quality BVHs can also be constructed automatically. One of the most used methods is *binned BVH construction* [37], published in 2007 by I. Wald. Their algorithm shares many similarities with their KD-tree building algorithm, published a year earlier [38]. A basic BVH construction algorithm is shown in Listing 2.2. It works by first generating a root node and then recursively subdividing that root node into two child nodes. Listing 2.2 shows allocation of the node pool and keeps track of the head of that pool using the *poolPtr* integer. First, the root node is created which is then subdivided into two child nodes. The *Partition* function determines which polygons end up in which child nodes.

```
void ConstructBVH(Polygon* polygons int polygonCount)
{
    //Allocate BVH node pool
    nodePool = new BVHNode[polygonCount];

    BVHNode root = nodePool[0];
    root.PrimitiveCount = polygonCount;
    root.IsLeaf = false;
    root.Bounds = ComputeBounds(polygons);

    //Set the head of the node pool pointer to two.
    poolPtr = 2;

    //Start recursively subdividing.
    Subdivide(root);
}

void Subdivide(BVHNode node)
{
    if(node.PrimitiveCount < 3)
    {
        //This node is marked as a leaf node.
```

```
            node.IsLeaf = true;
            return;
        }
        else
        {
            node.LeftChild = nodes[poolPtr++];
            node.RightChild = nodes[poolPtr++];
            Partition(node);
            Subdivide(node.LeftChild);
            Subdivide(node.RightChild);
            node.IsLeaf = false;
        }
}
```

LISTING 2.2: Basic BVH construction

Partitioning strategies work by selecting a plane along which the polygons in the current node are seperated. This seperation results in two new groups of polygons, that each will make up a child node. This seperating plane or *split plane* can be determined using the *surface area heuristic* (SAH) [27]. The SAH is a heuristic that can be used for evaluating the quality of a proposed split. It is formulated as follows:

$$SAH = A_{left} * N_{left} + A_{right} + N_{right} \tag{2.1}$$

The goal is to find a split plane that minimizes this heuristic as the heuristic is an indication for ray tracing performance. In the formula above, A indicates the area of the bounding volume of either the left or right groups of polygons that are separated by the split plane. This is multiplied by the number of polygons on either side of the split plane, represented in the formula by N.

Binned BVH construction as presented by I. Wald [37] considers a fixed number of split plane positions in the partition step, spaced equidistantly apart from each other. This results in a fast BVH construction algorithm, with high-quality BVHs, even though better split plane positions may lie in between the considered positions.

Many BVH construction algorithms exist that aim to construct higher quality BVHs while reducing computation time. For example, Bonsai [15] is a multi-threaded implementation that builds tiny BVH trees and groups them together afterwards. The SBVH [33] is a solution that does not just group consider a fixed number of split plane, but also allows splitting up large polygons during BVH construction. Many other building strategies exist, each with their own advantages and disadvantages. Furthermore, an MBVH can be constructed from a BVH by collapsing two levels of a BVH iteratively.

2.3 Global Illumination and Shading

Using rasterization or ray tracing, we can determine primary visibility of polygons and where their projection lies on the screen plane (Section 2.1). After this has been determined, the color of the pixel on the output image needs to be computed. If the goal of the rendering is to produce a realistic looking image, the final color of the output pixel depends on both properties of the material of the rendered surface point as well as the total, or *global illumination* that arrives at the surface point. The impact of shading and global illumination can be observed clearly in Figures 2.7 and 2.8.

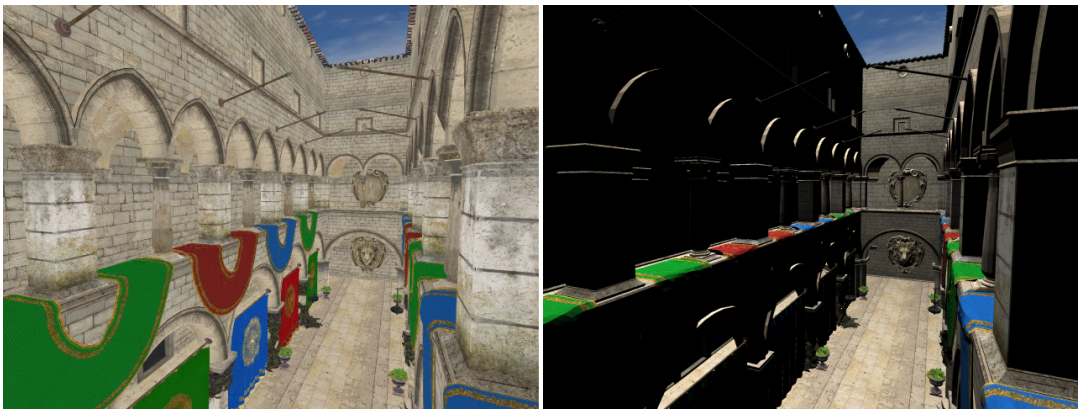

FIGURE 2.7: A render of the Sponza scene without (left) and with (right) shading.

In this section, we first show how global illumination at a surface point can be modelled accurately, using the rendering equation [24] in Section 2.3.1. Then, several rendering methods for global illumination are discussed. In Section 2.3.2, a *Monte-Carlo* algorithm known as *path tracing* is considered. This algorithm casts many rays to determine real-time global illumination. Then, Section 2.3.3 discusses *photon mapping*, which preprocesses the scene by shooting photons from the light sources and storing their surface interactions. Finally, materials and shading are considered briefly in Section 2.3.4.

2.3.1 Rendering Equation

The rendering equation, introduced by Kajiya [24], describes how light and materials affect the visual appearance of a surface point. It allows us to compute the amount of light leaving a surface point towards the observer, given the incoming light energy and material properties of the surface. From the rendering equation, a rendering system can be contrived that accurately renders photorealistic images.

The rendering equation can be written as shown in Equation 2.2 below.

$$L_o(\mathbf{x}, \omega_\mathbf{o}) = L_e(\mathbf{x}, \omega_\mathbf{o}) + \int_\omega f_r(\mathbf{x}, \omega_\mathbf{i}, \omega_\mathbf{o}) L_i(\mathbf{x}, \omega_\mathbf{i})(\omega_\mathbf{i} \cdot \mathbf{n}) d\omega_\mathbf{i} \tag{2.2}$$

with

- $L_o(\mathbf{x}, \omega_\mathbf{o})$ is the total outward radiance along direction ω_o from position x.
- $L_e(\mathbf{x}, \omega_\mathbf{o})$ is directly emitted radiance from position x along direction ω_o.
- $f_r(\mathbf{x}, \omega_\mathbf{i}, \omega_\mathbf{o}) L_i(\mathbf{x}, \omega_\mathbf{i})$ is the *bidirectional reflectance distribution function*, which is material property that specifies how much light from incoming direction ω_i is reflected towards outgoing direction ω_o at position x.
- $L_i(\mathbf{x}, \omega_\mathbf{i})$ is the amount of incoming radiance at position x from direction ω_i.
- $\omega_\mathbf{i} \cdot \mathbf{n}$ is the representation of light weakening due to incident angle. $\mathbf{n}$ represents the surface normal at position x.

Each term in the equation constitutes a specific portion of what makes up the radiance leaving a surface point. The first term, L_e, specifies directly emitted light

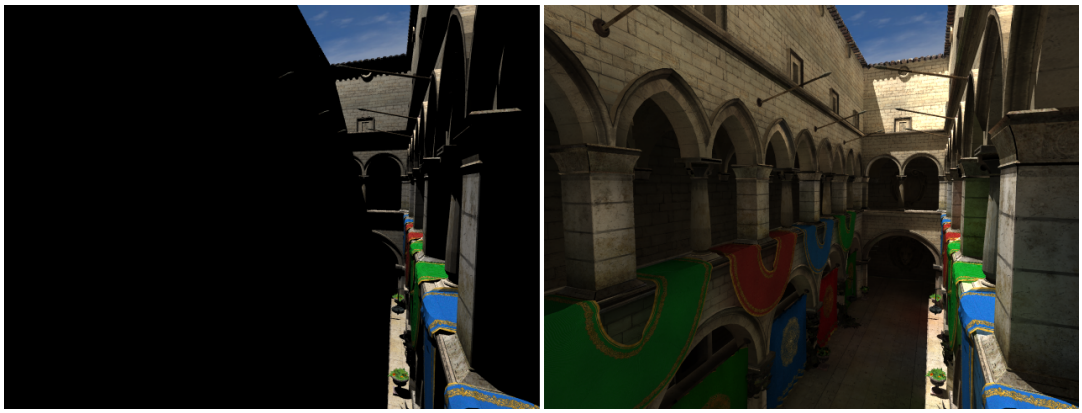

FIGURE 2.8: The same rendering of the Sponza scene without (left) and with (right) global illumination. Note that shadows have been added as well.

towards the observer. This will only contribute to the total outgoing radiance L_o of the surface point, if this surface is emmisive (ie, is a light source).

The integral in the equation is taken over the hemisphere centered around surface position $\mathbf{x}$ and oriented along surface normal $\mathbf{n}$. All terms within the integral represent how light arriving through this hemisphere contributes to the total outgoing radiance. Specifically, this is determined by the *BRDF* and the weakening factor of incoming irradiance when it is converted to radiance. Light may arrive through any direction over the hemisphere, either coming directly from light sources or arriving via bounces over other reflective materials.

The purpose of a rendering system is to evaluate the rendering equation. In Section 2.3.2, we describe the path tracing algorithm, which evaluates the rendering equation stochastically. Photon mapping, discussed in Section 2.3.3 tries to solve (part of) the rendering equation by precomputing light paths in the scene and storing their surface interactions.

2.3.2 Path Tracing

Path tracing is an algorithm that tries to approximate the integral over the hemisphere presented in the rendering equation in Section 2.3.1. It is an algorithm that builds on *distributed ray tracing* presented by Cook et al. [6]. Distributed ray tracing uses multiple rays to approximate a range of natural phenomena that can be modelled as an integral, such as depth of field and motion blur. Path tracing was presented in the same paper as the rendering equation by Kajiya [24].

Like traditional Whitted-Style ray tracing (see Section 2.1.2), intersection points with geometry are found using ray/polygon intersection tests. However, where traditional ray tracing then ends (or spawns additional extension rays), path tracing *always* spawns a new ray at the intersected surface point in a random outgoing direction. This random reflecting occurs at every subsequent surface point that is intersected, until the ray intersects a surface that emits light. When it does, the path is terminated and we have found a path along which light is transported from the light source, to the observer. Using this path, we can compute exactly how much light energy is transported from the light source, to the observer. This process is shown in Figure 2.9, where a ray from the observer bounces twice until it finds a light source and then terminates.

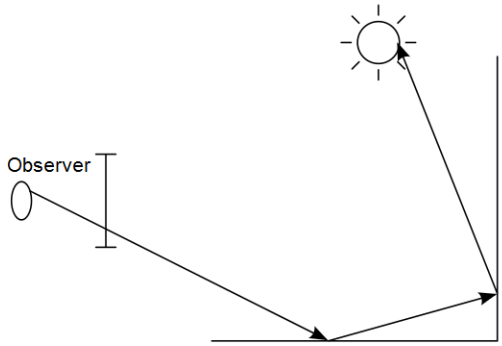

FIGURE 2.9: An example of a path along which light can be transported towards the observer. The path is traced in reverse, from the observer to the light source, avoiding computation of many useless paths that would never reach the observer.

In a single iteration of path tracing, we typically shoot one ray through each pixel in the screen plane, just like in traditional ray tracing. The result is a noisy image, because light may arrive through many paths towards the observer, and only one might have been found per pixel. By averaging many output frames together, the noise is reduced, as each frame represents different paths that light may take towards the observer. This process of *convergence* can be observed in Figure 2.10.

Because many paths must be found before the image converges, path tracing is not very suitable for use in real-time applications. However, path tracing is used in experimental games [19] and interactive physically based renderers such as KeyShot [20]. By applying variance reduction techniques and implementing fast ray traversal structures, these applications make real-time path tracing a possibility.

Bidirectional Path Tracing

Bidirectional path tracing by Lafortune et al. [25] attempts to make convergence (see Section 2.3.2) faster by tracing light paths from both the light sources and the camera. A path from the light source into the scene is traced independently from the path from the camera. These two are then connected with each other, to obtain faster convergence. While this takes extra computations for each path, convergence is speed is much higher and thus, the final image is formed earlier.

Connection of a camera path to a light path is achieved by casting shadow rays from the camera path intersection points to the light path intersection points. For determining which paths are connected to which, importance sampling can be used as described by E. Veach [36].

2.3.3 Photon Mapping

Instead of determining global illumination live, *photon mapping*, introduced by H.W. Jensen [22] in 1996, uses a preprocessing step to determine light distribution across the scene before rendering images. The algorithm is thus a two-pass algoritm, consisting of two phases. First, the *photon tracing phase* (Section 2.3.3) traces photons from the light sources into the scene to determine the photon distribution. Then, the

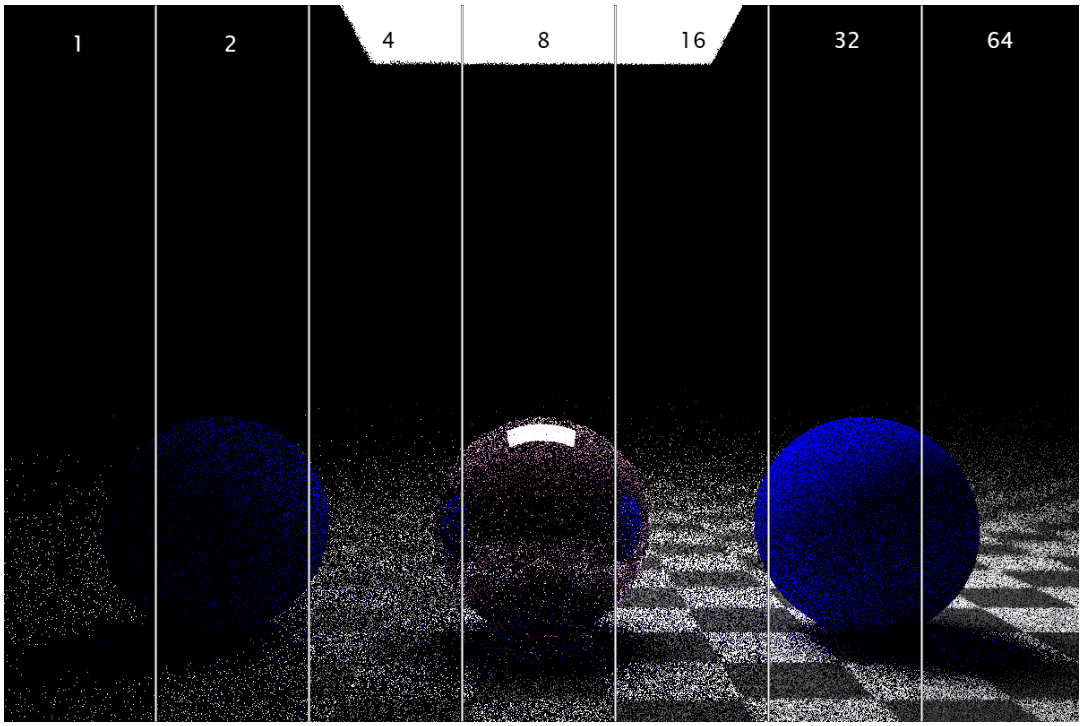

FIGURE 2.10: As the number of samples (shown above) increases, the noise is reduced.

rendering phase uses the photon distribution to determine a radiance estimate that can be used to render global illumination in the final image.

Photon mapping has been used in real-time for global illumination on the GPU [29, 10]. Furthermore, higher visual quality of the output image can be obtained by applying techniques such as progressive photon mapping [17].

Photon Tracing

During photon tracing, a fixed number of photons is emitted from the light sources. These photons bounce around the scene, similar to how paths in path tracing (Section 2.3.2) bounce. Each indivual photon path segment can be modelled as a ray. When a photon interacts with a surface, this interaction is stored in the *photon map*. The photon map is a datastructure that contains all interactions of all photons with surfaces in the scene.

After the interaction is stored, a technique known as *Russian Roulette* is used to determine what will happen to the photon [23]. This is a chance experiment that determines whether the photon continues along its path and bounces around, or whether the photon is absorbed by the surface. After the photons have been traced through the scene, it is useful to store the interactions in a manner that they can easily be retrieved given a position. H.W. Jensen advocates the use of a KD-tree for this purpose. For more information on spatial datastructures, see Section 2.2.

To further speed up the photon tracing phase, we may choose to emit more photons from bright light sources or use projection maps [23] to emit photons towards geometry.

Radiance Estimate

After the photon tracing phase is complete, the rendering phase begins. To determine the lighting at a surface point, we can either use the photon map to directly determine a radiance estimate, or we can use *final gather*, discussed in Section 2.3.3. A direct visualization of the radiance estimate of 10M photons can be seen on the left in Figure 2.11.

To use the photon map, we must compute the *radiance estimate*. The radiance estimate at a surface point can be computed by gathering nearby photon interactions and adding these together [23]. Specifically, photons can be gathered in a sphere surrounding the surface point, with an expanding radius. Once a predetermined number of photons is included in the sphere, the search stops and the radiance estimate can be determined. This may result in photons being included in the estimate that should not have been. For example, near corners, photons that lie on both sides of the corner are included. This can be partially solved by using a disk instead of a sphere, however, thin walls and other obstacles may still cause errors. This is mostly solved by *filtering*, discussed below.

Filtering Using a filter, we can create a radiance estimate that takes into account photons according to their distance from the surface point. Using this strategy we can directly compute the outgoing radiance estimate on a surface point based on the distribution of photons around that surface point. With a high resolution photon map, this can produce good-looking results. Several filters have been proposed by Jensen et al. [22] with filters specifically suited for caustics as well as indirect illumination.

Final Gather

Final Gather (FG) is a technique to improve the visualization of indirect illumination at a surface point using a photon map. Where direct visualization of the radiance estimate results in very blurry illumination, using final gather, we can get very accurate and smooth global illumination.

Final gather works by casting a number of additional rays from the primary ray intersection points. Just like path tracing, these extended rays are used to gather illumination along the light path. However, where path tracing paths continue until a light source is found or the ray leaves the scene, final gather paths typically terminate at the first surface intersection. At this FG intersection point, the photon map is queried to determine the radiance estimate at the FG intersection point. Light transport to the original primary ray intersection point is computed and this is averaged together over all the FG rays spawned at the primary ray intersection point. This results in smooth indirect illumination at the rendered surface point. Using traditional light sampling, direct illumination can be determined and the sum of these is the total illumination at the surface point.

The effects of adding final gather to a renderer can be seen on the right in Figure 2.11. While final gather is an approximation and not entirely physically correct (due to the limited accuracy of the photon map), the method is much faster than computing complete paths using path tracing.

Light Baking

Like object color can be stored in a texture map, light intensity at a surface point can be stored as well. Such maps are referred to as *light maps* and the process of creating

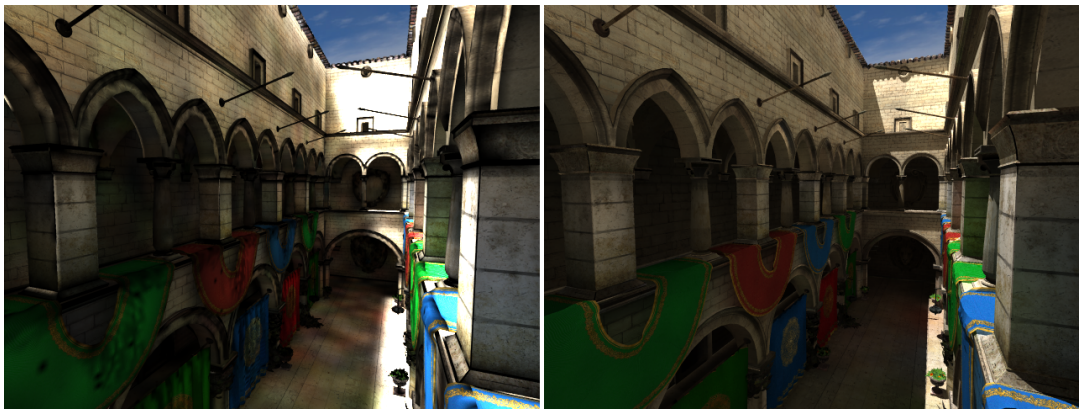

FIGURE 2.11: A direct visualization of the radiance estimate at the surface points (left), compared to a visualization where the radiance estimate is only used for indirect lighting (right) using *final gather*.

such light maps from lighting data (such as a photon map) is called *baking*. The lighting data is essentially baked into a map, so that it can be retrieved at rendering time using UV-coordinates. At baking time, the UV-coordinates can be used as a parametrization over the surface of the models. Because lighting is unique all over the object, all UV-coordinates in the UV-map must also be unique. Therefore, many pre-existing UV-mappings over a model cannot be used and a new mapping must be generated.

During baking, it must first be determined which UV-coordinates correspond to which triangle coordinates. This preprocessing step builds a 2D map that stores pointers to the triangles and their corresponding world space coordinates. Then, during baking, all UV-coordinates are iterated over and their corresponding world space positions are retrieved. For each world space position, the radiance estimate is determined using the photon map. This radiance estimate is stored in the light map at the original UV-coordinates. Then, the radiance estimate light map can either be visualized directly or be used in a final gather renderer. Final gather rays can also be spawned at the world space coordinates corresponding to the UV map, and this lighting in turn can be baked into a seperate light map, to be visualized directly [4].

2.3.4 Materials

The appearance of a surface point depends on the way light interacts with it, which can be modelled using a *bidirectional scattering distribution function*, or *BSDF*. Surfaces are always both slightly transmissive as well as reflective. Often, the transmissive component is ignored completely in rendering, in which case we are left with the *bidirectional reflectance distribution function*, or *BRDF*. The BRDF determines the ratio of light that is reflected by a surface point, given an incoming light direction and outgoing light direction.

Diffuse surfaces can be rendered by using *Lambertian* shading. Perfectly diffuse surfaces scatter light equally in all directions across the hemisphere centered around a surface point and oriented around the surface normal. Therefore, the diffuse BRDF is constant across the hemisphere.

Glossy surfaces differ from reflective surfaces as they do not scatter light precisely along the reflected direction of an incoming light ray. The difference is clearly visible in Figure 2.12.

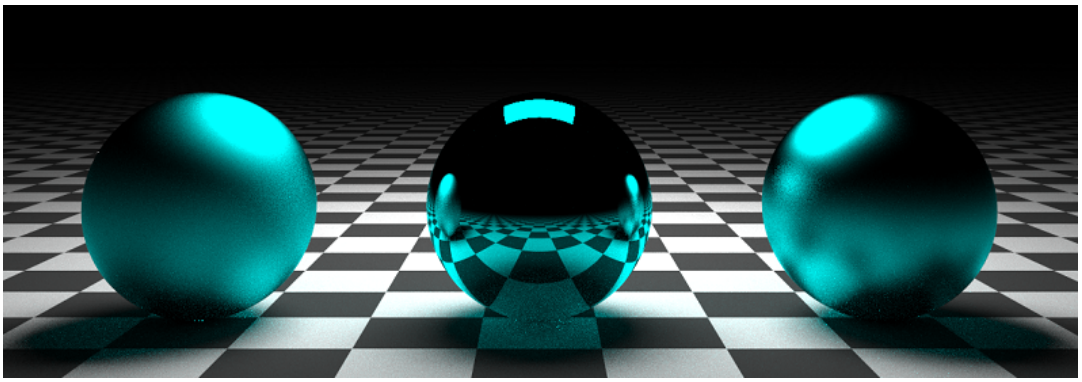

FIGURE 2.12: Three spheres, each with a different surface material.

Phong shading [31] is an implementation of a BRDF for glossy surfaces. Many other, physically-based BRDFs exist, such as the microfacet model by K.E. Torrance and E.M. Sparrow [35]. Accurately rendering these BRDFs requires an integration step over the hemisphere at the visualized surface point, because the material may scatter light in any of these directions. However, an approximation can be rendered that only takes into account direct illumination from (point) light sources in the scene.

Chapter 3

Implementation

In this chapter, the implementation of our pipeline for precomputed global illumination is presented. The key ingredients are the acceleration structures for ray tracing, the global illumination pipeline and our live updates to the light maps. The acceleration structures are used for both rendering and photon path tracing. Our global illumination pipeline consists of building a photon map and deriving a light map. Using live updates, we can construct a coarse photon map quickly and incrementally increase detail during rendering.

Our primary goal is to create an extensible framework that allows for precomputing global illumination and visualizing the results. The approach we present is not new. Light map baking can use any illumination model to determine the lighting over the surface of a mesh. In our case, we have chosen to use a photon map because of its fast construction time and flexibility. A similar approach is mentioned by Christensen [5].

Section 3.1 presents our implementation of BVH and MBVH construction and traversal algorithms. Then, in Section 3.2, our global illumination pipeline is discussed. Finally, our implementation of live updates to the photon map is presented in Section 3.3.

3.1 Ray Traversal Acceleration Structures

Our implementation of photon mapping and final gather relies heavily on the ability to quickly trace rays and find their intersection with scene geometry. Furthermore, our scene is rendered using ray tracing. In accordance with our findings in the literature in Chapter 2, we have chosen to implement the BVH for primary ray traversal and the MBVH (Section 2.2.3) for incoherent ray traversal. Our implementations are discussed below in Sections 3.1.1 and 3.1.2.

3.1.1 BVH

We have implemented the binned BVH construction algorithm as described by I. Wald [37] for fast BVH construction. We have seperated traversal and construction of the BVH, which allows the same BVH to be traversed by multiple traversal algorithms. Two traversal strategies have been implemented for the BVH. First, we have implemented iterative, ordered single ray traversal (see Section 2.2.3). This is mostly used for debugging purposes, considering that single ray traversal using the MBVH is faster. Packet traversal according to Overbeck et al. [30] has been implemented as well, which is used to trace primary rays for rendering. All elements of the original papers, for both construction and packet traversal, have been transferred to our implementation. Performance of all traversal strategies is reviewed in Section 4.1.

3.1.2 MBVH

Our MBVH construction algorithm consists of collapsing an existing BVH. Each node in the BVH adopts its child nodes' children, to a maximum of four children per node. In this manner, each node in the MBVH can contain up to four children. The algorithm is analagous to the BVH collapse algorithm by Dammertz et al. [9]

We have adapted our single ray traverser for the BVH to work with the MBVH structure. The adjustment is trivial: at each interior node, the four child nodes are checked and if they intersect with the ray, pushed on the stack. Ordered traversal is not implemented here. This is our fastest option for incoherent, random ray traversal (see Section 4.1).

3.2 Global Illumination Pipeline

Our global illumination pipeline is capable of precomputing diffuse illumination on large scenes. It has been tested on scenes containing up to twelve million triangles.

Our global illumination pipeline is capable of computing diffuse global illumination for perfectly diffuse surfaces. Multiple light sources are supported. These can be both point lights, as well as rectangular area lights. The global illumination pipeline does not handle glossy or specular surfaces, nor does it handle caustics or transmissive objects.

In this section, the global illumination pipeline is decomposed into several components, which are then discussed seperately. An overview of the pipeline is given in Section 3.2.1. Each individual component is discussed in turn, in Sections 3.2.2 through 3.2.6.

3.2.1 Implementation Overview

An overview of the steps in the pipeline will be given here, with each step being worked out in more detail in subsequent sections. The implementation consists of the following steps:

1. *Photon Map* - A photon map with a uniform grid for fast lookup.
2. *UV Mapping* - A parametrization of the mesh is created in order to enable light map baking.
3. *Radiance Map* - The photon map and UV map are used to create a radiance map, a parametrized representation of the global illumination created by computing the radiance estimate over the surface of the mesh.
4. *Final Gather Map* - The final gather map is created by tracing rays at discretized points of the UV map. The results are stored in an indirect illumination map.
5. *Light map* - The light map is a parametrized representation of the global illumination in the scene. It combines indirect illumination from the final gather map with deterministically computed direct illumination.

The following sections will discuss each step in the pipeline in more detail.

3.2.2 Photon Map

Our implementation of the photon map is straightforward and mostly according to the practical guide to photon mapping by Jensen et al.[23]. First, photons are emitted from the light sources and traced through the scene. They are stored for quick lookup in a uniform grid, discussed below. The radiance estimate is determined using a cylinder and cone filter.

Emission

During the emission phase, a fixed number of photons are emitted from randomly selected light sources, in a random direction over the hemisphere. The photons are traced through the scene using the MBVH traversal algorithm discussed in Section 3.1.2, as this appears to perform best for incoherent ray sets (see Section 4.1).

Our photon datastructure consists of a photon position, direction and power. At each surface interaction, the position of the photon is stored, along with its power. Then, using Russian Roulette as described in [23], it is determined whether the photon is absorbed or reflected. If it is reflected, the photon power is reduced according to the material properties of the surface and the process repeats. In this manner, a distribution of photons throughout the scene is determined, with photon density corresponding directly to light intensity. In other words, at brightly lit areas, the number of photons present is larger than in dimly lit areas.

For performance reasons, our implementation has a configurable maximum number of bounces that a photon may undergo along its path. Furthermore, our photon tracing implementation is multithreaded and utilizes SIMD instructions to speed up photon tracing.

Storage and lookup

After the distribution of photons in the scene has been determined, they are organized in a uniform grid (see Section 2.2.1). The uniform grid can be constructed rapidly and allows for much faster lookup times. During construction, photons are placed in a grid cell based on their position in the scene. The grid has been chosen as our photon map acceleration structure for its relative simplicity to implement. In the future, the grid may be replaced by other acceleration structures, if these prove to be more efficient in lookup or construction.

Radiance estimates can be determined at any surface point in the scene by querying the photon map. The photon map then uses the uniform grid to search linearly through the grid cell which contains the surface point, as well as its neighbours which fall within the search radius.

Radiance Estimate

The radiance estimate at any surface point can be determined using the photon map and the grid. Our implementation uses a cylinder to determine which photons should be included in the radiance estimate. The photon map is queried by providing the surface point, a cylinder radius and cylinder height. All grid cells which overlap with the sphere around the surface point are searched. Only those that fall within the cylinder are included in the radiance estimate.

During the search, all photons within the searched grid cells are tested. If the following two conditions are met, the photon falls within the search cylinder and is included in the radiance estimate:

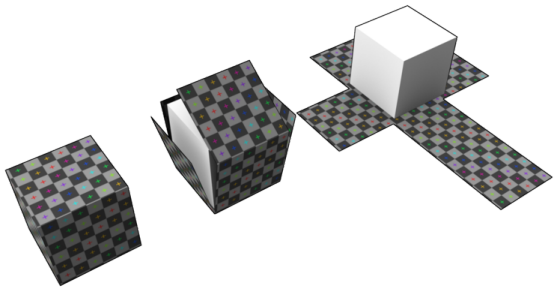

FIGURE 3.1: A depiction of the unwrapping process of a cube. Each point in the flattened cube corresponds to a single point on the surface of the 3D cube. Attribution: Zephyris at en.wikipedia.

1. The distance from the photon to the surface point is less than the search radius.
2. The distance from the photon to the plane oriented perpendicularly to the surface normal at the surface point is less than the cylinder height.

Once all photons that meet these two conditions have been located, the radiance estimate is determined by summing their power and normalizing the result using the cone filter described by Jensen et al. [23]. Note that our implementation differs from theirs as we use a volume of fixed size (the cylinder) for the photon search, rather than an expanding volume. Where the original algorithm stops when a fixed number of photons have been located, ours stops when all photons within the volume are located. Our approach makes performance more predictable, as there is a maximum number of grid cells that can be considered during the search, while maintaining quality of the estimated lighting. Furthermore, we can tune the size of the grid cells to the fixed search radius.

3.2.3 UV Mapping

The final output of our global illumination pipeline is a light map. This is a representation of the light in the scene that is two-dimensional and depends upon a parametrization of the surface of the scene. This parametrization is referred to as a UV-map. A UV-map maps each three-dimensionsal surface point to a two-dimensional point in UV-space. The process of determining a UV-map is called *unwrapping*. Figure 3.1 visualises a UV-map of a cube, with textures overlayed on both the cube and the UV-map.

In our case, the UV map needs to be unique: each point on the UV map must correspond to exactly one point on the surface of the 3D scene. We will be storing the lighting in a texture, based on the UV map. This one-to-one mapping is required because the lighting in the scene is not repetitive and differs at every surface point.

Splitting

Unwrapping a 3D model is a whole other area of research. Therefore, we have chosen to use a third-party SDK to perform the unwrapping of arbitrary scenes for us.

FIGURE 3.2: The radiance map visualized directly on the Sponza scene (left), containing global illumination. A visualization of the *final gather* map, containing indirect illumination only (right).

Microsoft prodives such an SDK in the form of *UVAtlas*. Their SDK becomes increasingly slow as the number of polygons in the scene increases, making it nearly impossible to unwrap models of several millions of polygons. To counter this effect, we have chosen to split the model up into parts and then feeding these parts to UVAtlas. This allows UVAtlas to scale (close to) linearly in the number of parts. This requires multiple UV maps, radiance maps, FG maps and light maps, thus taxing memory heavily. Fortunately, each map can be of a lower resolution than when a single map is used for the entire scene. Another advantage is that each part of the model can be unwrapped concurrently, allowing for a multithreaded implementation.

Inverse Mapping

A UV map allows us to go from a 3D surface point in world-space to a 2D point in texture space. In our case, we will often find that we need to go from 2D texture space, to 3D surface points. This will become necessary to determine illumination over the surface of the mesh. Therefore, an inverse mapping must be constructed.

We determine the inverse mapping by looping over the unwrapped triangles. For each triangle, we determine the UV coordinates of it's three vertices. Then, for each UV-space pixel in the bounding rectangle of the three UV coordinates corresponding to the vertices, we check whether it is contained within the UV-space triangle formed by its three UV coordinates. If this is the case, we can pair the world-space triangle and the current UV-space pixel.

At runtime, we can compute the barycentric coordinates of any UV-space point by looking up the triangle corresponding to the nearest whole UV-space pixel. These barycentric coordinates can then be converted to world-space using the world-space coordinates of the three vertices.

3.2.4 Radiance Map

After the photon map has been computed and a UV parametrization is available, the radiance map can be constructed. The radiance map is a texture that contains the radiance estimate at each surface point of the mesh. It uses the inverse unique

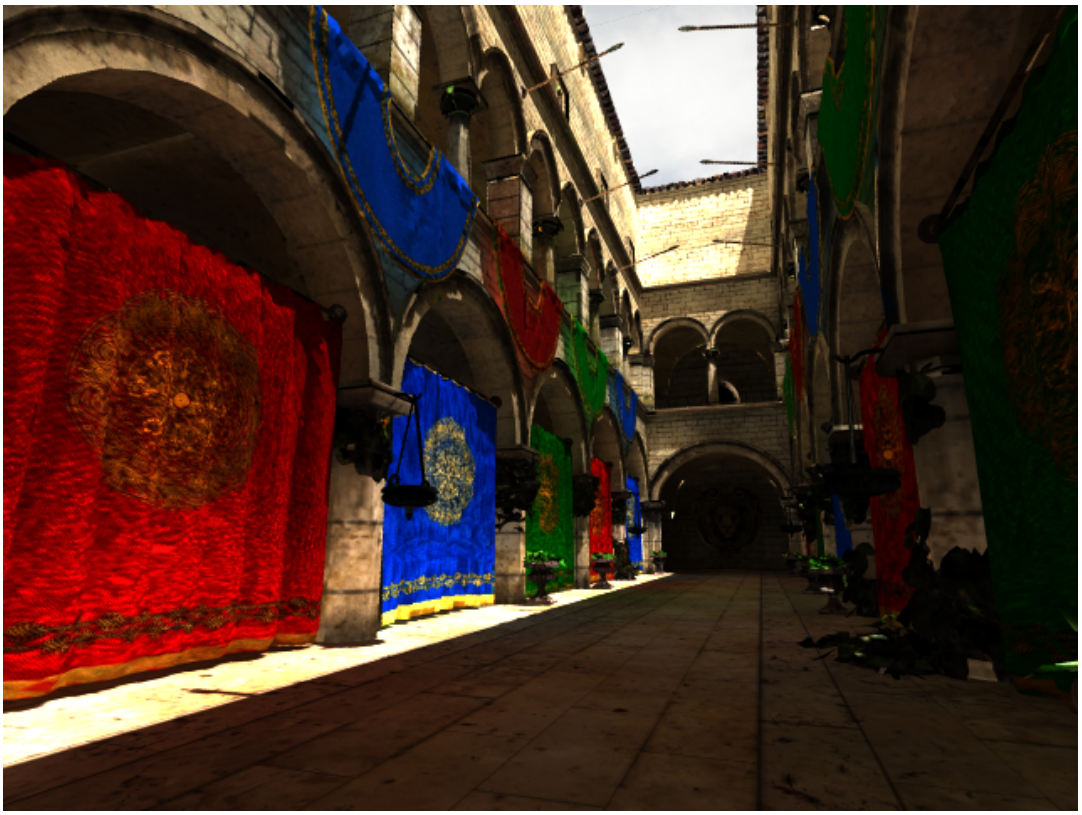

FIGURE 3.3: The full light map visualized. The light map has a resolution of 4096x4096 pixels, to allow for crisp shadows.

mapping constructed using UVAtlas (Section 3.2.3). The radiance map can be constructed by determining, for each pixel in the inverse mapping, the radiance estimate that corresponds to the world-space coordinate in the scene.

In this manner, a texture map containing the radiance estimate over the entire scene is constructed and stored for use in the next step of the pipeline. A visualization of the radiance map can be seen for the Sponza scene in Figure 3.2 on the left. Our implementation is multithreaded.

3.2.5 Final Gather Map

The radiance map contains an estimate of the global illumination at each surface point in the scene. In our approach, we want to seperate indirect illumination from direct illumination, as the latter can often be computed rapidly in a deterministic manner (especially for point lights).

When constructing the final gather map, this is exactly what happens. As discussed in Section 2.3.3, final gather works by casting aditional rays from the primary ray intersection points into the scene. At each FG ray intersection, the radiance estimate is determined using the photon map. Light transport back to the original primary ray intersection point is then determined and used to shade the point. In our case, we want to build a map using final gather, that maps the indirect illumination over the surface of the scene.

To this end, we use the inverse mapping constructed using UVAtlas (see Section 3.2.3) to iterate over the surface. For each UV-space pixel, we compute the corrensponding world-space coordinate and use this as the origin for a predetermined

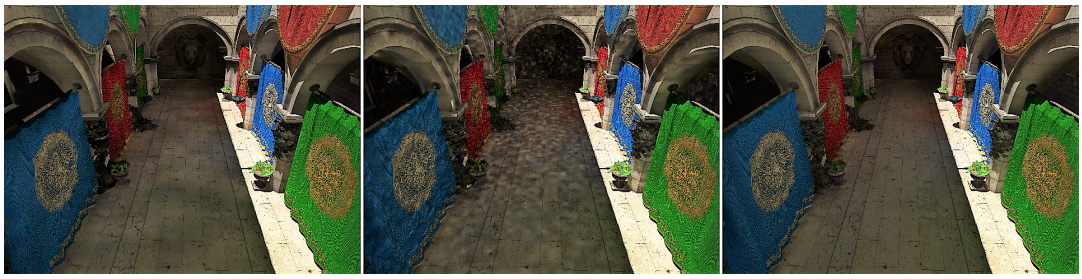

FIGURE 3.4: A visualization of the view-based updates. The left image shows initial lighting, computed using eight FG rays. The middle image shows lighting being updated by casting additional FG rays. The right image shows the converged result, as enough FG rays have been cast for detailed lighting. The image has been brightened for the purpose of comparison.

number of final gather rays. At each final gather ray intersection point, we do not determine the radiance estimate using the photon map, but instead pull it directly from the radiance map constructed according to Section 3.2.4. The resulting light transport from the final gather rays is indirect illumination only and is stored in a texture. After the final gather map is complete, the map is blurred with a small, adjustable blur kernel to smooth out the soft indirect illumination. The final gather map is visualized in Figure 3.2 on the right.

Like photon mapping, final gather traces a large number of rays through the scene. Like photon path tracing, the most efficient traversal strategy is MBVH traversal, as described in Section 3.1.2. Our implementation is, once again, multithreaded.

3.2.6 Light Map

The final step in our pipeline is to determine the complete light map. The light map combines indirect and direct illumination into a single texture. Its size is independent of the size of the final gather map. To construct the light map, direct illumination is determined at each surface point corresponding to each UV-space pixel in the inverse mapping. Then, indirect illumination of the closest final gather map pixel is added. The result is global illumination at the surface point, stored in a texture map. A full light map visualization can be seen in Figure 3.3. The visualized light map has a resolution of 4096x4096, allowing for crisp shadows.

3.3 Incrementally Updating the Light Map

As discussed in Chapter 1, our global illumination pipeline will be used to preview megastructures (massive scenes) by architects and designers, during their design process. This means that startup time of the visualization application must be short as it is undesirable to wait for prolonged periods of time between visualization of the megastructures. Application startup time can be decreased by lowering the light map resolution. This reduces the quality of the baked lighting, as the resolution of the map is now lower. At runtime, a higher resolution version of the light map can be improved in the background, incrementally accumulating detail in the lighting. The light maps can then be switched out once enough detail is present, gradually increasing the quality of the lighting in the scene.

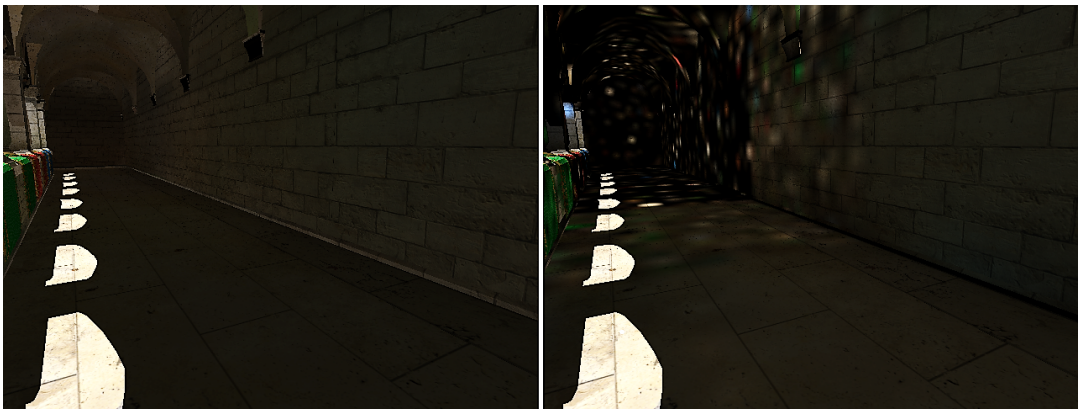

FIGURE 3.5: A visualization of the position-based updates. The left image shows initial lighting, computed using sixteen FG rays. This is what the user sees at runtime. The right image shows what is going on in the background: additional FG rays are cast at higher LOD levels to compute more detailed lighting. The image has been brightened for the purpose of comparison.

In this section, two light map updating techniques are discussed. The first technique initializes high-resolution maps during start-up, but computes lighting sparsely on these maps. The resulting lighting is blurred out to fill up the light map. At runtime, view-based updates are applied to the light map. This technique is discussed in Section 3.3.1. The second technique uses a hierarchy of light maps and a uniform grid to update lighting based on the position of the camera. This position-based technique is discussed in Section 3.3.2.

3.3.1 View-based Updates

Our must rudimentary method of updating the light map is the view-based updating technique. At application start-up, the light maps are initialized at the final, desired resolution. However, during baking, not every pixel of this light map is considered. In essence, a lower resolution light map is computed and scaled-up to the final desired resolution. This leaves lighting coarse at start-up.

At runtime, our raytracer traces primary rays from the camera, into the scene. At the intersection points of these primary rays with the scene geometry, additional final gather rays are spawned using a chance experiment. The closer a primary ray is to the center of the screen, the higher the probability of spawning a final gather ray at its intersection point. These final gather rays are traced within the same frame, using the result to update the light map. This is visualized directly, resulting in spotty lighting, as can be seen on the left in Figure 3.4. However, after the camera has viewed the same area for a while, the light map starts to converge, resulting in more detailed lighting. This can be seen on the right in Figure 3.4. Furthermore, it is possible to blur the lightmap using a small blur kernel, to make the lighting appear smoother.

3.3.2 Position-based Updates

The position-based updating technique is more sophisticated than the view-based updating technique discussed in Section 3.3.1. The position-based updating technique consists of three components, each discussed in more detail below:

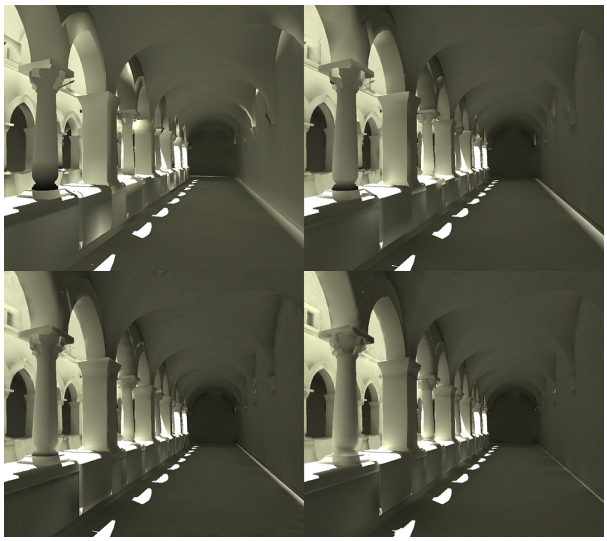

FIGURE 3.6: A visualization of the different LOD levels in the light map hierarchy. The left-upper image has a light map resolution of 256x256, the right-upper 512x512, left-lower 1024x1024 and the right-lower 2048x2048.

1. Light Map Hierarchy - A hierarchy of light maps, doubling in resolution at each level of the hierarchy.
2. Light Map Grid - A grid that divides the light maps up in cells. These cells are used to update the light maps.
3. Automatic LOD switching - As light maps are updated, higher LODs are switched to automatically, once they are of sufficient quality.

Our system allows us to update light maps based on the position of the camera. Cells of the grid that are close to the camera are updated more frequently than those further away, resulting in higher quality lighting near the observer. Furthermore, the system hides the update process from the user: the updates are applied to the light maps in the background. The updated light maps are only visualized once enough detail has accumulated. Using this system, high-quality lighting can be rendered at runtime and combined with a fast application start-up.

Light Map Hierarchy

Instead of initializing the light maps at their final, desired resolution, a hierarchy of light maps is initiliazed. Each level in the hierarchy contains a light map and is referred to as a light map LOD (level of detail). As we move up one level in the hierarchy, the resolution of the light map is doubled. At application start-up, the lowest light map in the hierarchy is computed entirely. The other levels of the light maps are updated using the light map grid (Section 3.3.2) and switched to automatically once they are detailed enough according to our automatic LOD switching (Section 3.3.2). Several levels of the light map hierarchy are visualized in Figure 3.6. While returns are diminishing, clear differences are present between each level of the light map hierarchy.

FIGURE 3.7: Two grid cells near the camera have switched over the next level of detail. A border is visible between the grid cells that are at the lower LOD level and those that are at the higher level. The image has been brightened for the purpose of comparison.

Light Map Grid

The light map grid is a uniform grid of tuneable resolution, that is overlayed over the scene. The grid is constructed by looping over all UV-map pixels and determining which grid cell their corresponding world-space position is in. This then assigns that UV-pixel to the corresponding grid cell. In essence, each grid cell contains a partial light map. Furthermore, each grid cell keeps track of which LOD level it is at. During start-up, this is set to the lowest level for each grid cell.

At runtime, grid cells that are close to the camera are selected for updating. The grid cell has a set of UV-pixels associated with it and one is selected at random from this set. A final gather ray is traced from the 3D position corresponding to the randomly selected UV-pixel and the result is stored in a higher level of the LOD hierarchy. These updates are not yet visible: using the grid, a higher level of the LOD is thus updated in the background. The process of updating in the background is illustrated in Figure 3.5.

Automatic LOD Switching

As each cell of the grid keeps track of its own LOD level, the grid can be queried to determine which LOD must be rendered at which position in the scene. Once a primary ray intersects a polygon, the light map grid is queried to determine which LOD level must be used for shading.

At runtime, the light map pixels corresponding to the grid cells are updated as described in Section 3.3.2. Per grid cell, the number of updates is kept track of. Once a predetermined threshold has been reached, the grid cell applies a slight blur to the partial light map it contains. Then, the grid cell increments its current LOD level and the next LOD is thus used for shading. A grid cell that has switched to the next LOD is visualized in Figure 3.7.

Chapter 4

Results and Validation

To determine whether our implementation of the global illumination pipeline is both sufficiently performant and produces high-quality results, we have set up several experiments. In this chapter, each experiment is first described, its results presented and then discussed. We will refer to such a trio of setup, results and discussion as an *evaluation*. Each section or subsection of this chapter adheres to this structure and presents a single evaluation.

The scenes which we use in our experiments are the well-known *Sponza*, *San Miguel* and *Powerplant* models. We have also used scenes that contain the Powerplant model four times, eight times and twelve times respectively. Sponza contains 262,267 triangles, while San Miguel and Powerplant contain 7,852,948 and 12,759,246 triangles. Powerplant x4, x8 and x12 contain 51,036,984, 102,073,968 and 153,110,952 triangles. All our measurements were taken on a system with an Intel Core i7 5930K processor, clocked at 3.50 GHz. The system had 64 GB of RAM available.

First, the overall quality of the BVH and MBVH construction algorithms as well as the traversal algorithms are evaluated in Section 4.1. After that, the performance of the global illumination pipeline is evaluated in Section 4.2. Its individual components are evaluated as well as the overall running time of the pipeline. Lastly, an evaluation of the visual quality of global illumination is presented in Section 4.3.

4.1 (M)BVH Performance Evaluation

In this section, we evaluate the performance of the key ingredients of the global illumination pipeline and rendering system, the BVH and MBVH. Both the construction time is evaluated in Section 4.1.1 and the traversal performance is evaluated for primary rays and random rays in Section 4.1.2. Our implementations are according to Sections 3.1.1 and 3.1.2. Note that BVH and MBVH traversal performance is also discussed extensively in the accompanying small paper, found in appendix A. All construction and traversal experiments in this section are single-threaded.

4.1.1 BVH and MBVH Construction Time

Experiment Setup

To determine whether the construction time of the BVH and MBVH is fast enough for the massive scenes consisting of several billion polygons, we have set up an experiment that tests the construction time of the BVH and MBVH, using the scenes described in the chapter introduction.

Our binned BVH builder uses 64 bins and has a maximum of three primitives in a leaf node. Using a stopwatch in code, the duration of the BVH construction process is measured and recorded. This process is repeated five times, after which

the resulting measurements are averaged together. The same is done for the MBVH construction process. The results are shown in Table 4.1.

Results

Scene	BVH construction time	MBVH collapse time
Sponza	0.983	0.059
San Miguel	35.35	1.694
Powerplant	62.23	2.824
Powerplant x4	269.5	12.60
Powerplant x8	968.3	30.79
Powerplant x12	1456.0	134.22

TABLE 4.1: BVH and MBVH construction times for several different scenes. All times are in *s*.

Discussion

The BVH construction time clearly dominates over that of the MBVH. Therefore, any future improvements should focus on constructing the BVH more rapidly. At the moment, our BVH construction implementation, according to Wald et al. [37] is fast enough for testing and supporting the global illumination pipeline.

Our implementation is straightforward and unoptimized, nor does it use SIMD. Therefore, if necessary, our implementation can be sped up. More likely, a different construction algorithm will be needed to support the massive scenes. We propose to use either the Bonsai BVH construction algorithm [15] or create an implementation of a parallel SBVH construction algorithm by Fuetterling et al. [14].

4.1.2 Ray Traversal Performance

Experiment Setup

As established in Chapter 3, implementing fast ray traversal algorithms is critical for achieving performant photon mapping and final gather. In this experiment, we test the performance of our ray traversal algorithm implementations. The experiment setup is the same for each experiment, varying either the BVH/MBVH, the traversal strategy or the ray type.

All experiments start by generating a batch of rays. In the case of primary rays, one hundred sets of primary rays for a screen of 640x640 pixels are constructed. This results in 640x640x100 rays. In the case of final gather rays, the total batch size remains the same, except rays are generated at random. The starting point of a ray is a random point on a randomly selected triangle within the scene. The direction of the ray is generated randomly in the hemisphere oriented along the triangle normal. All primary rays are spawned from the first viewpoints used in Section 4.3. After the ray batch has been generated, a code stopwatch is set and each ray or ray packet is traced in turn. After the ray batch traversal process has completed, our stopwatch time is measured. From this data, we compute the number of rays per seconds that are traced.

Results

Scene	Ray Packets (BVH)	Single Ray (BVH)	Single Ray (MBVH)
Sponza	3.330	1.078	1.030
San Miguel	1.898	0.989	0.943
Powerplant	2.252	0.373	0.389
Powerplant x4	2.027	0.289	0.293
Powerplant x8	1.902	0.240	0.247
Powerplant x12	1.691	0.228	0.237

TABLE 4.2: Primary ray performance for three traversal strategies. Expressed performance figures are in MRays per second.

Scene	Single Ray (BVH)	Single Ray (MBVH)
Sponza	0.281	0.291
San Miguel	0.093	0.097
Powerplant	0.071	0.072
Powerplant x4	0.058	0.060
Powerplant x8	0.049	0.050
Powerplant x12	0.046	0.049

TABLE 4.3: Final Gather ray performance for two traversal strategies. Expressed performance figures are in MRays per second.

Discussion

Primary Rays As can be seen from the results, the large packet traversal strategy clearly works best for primary rays, especially as scenes become larger. The results also show that both the geometrical layout of the scene and the camera position affect ray traversal performance. While the Powerplant scene is twice as large as the San Miguel scene, it still renders faster using packet traversal. This is likely caused by the fact that many ray packets completely miss the large, geometrically complex Powerplant structure. These can be culled efficiently using the frustum test described in Overbeck et al. [30]. Single ray BVH and MBVH performance are similar to each other.

The current traversal strategies are not fast enough to render the proposed megastructures in real-time at high-definition resolution. Fortunately, this is not a problem, as a rasterization solution is in place. In the future, primary ray traversal can be sped up to allow for such rendering. Our implementation is not optimized and does not utilize SIMD to test multiple rays against the BVH bounding boxes or polygons. Furthermore, algorithmic improvements can be made by implementing ray traversal according to Fuetterling et al. [13]. However, the focus for now was on creating an extensible framework upon which can be improved later on.

Final Gather Rays Our experiment for final gather rays shows that the coherence and starting point of rays can have a huge impact on performance. All our final gather rays have a different starting point and a different direction. This makes them extremely incoherent, as they can start nearly anywhere in the scene. Furthermore, as the starting point of the rays always lies on a triangle surface, the BVH traversal process will often reach deeply within the tree.

Final gather ray traversal performance is fast enough for precomputing global illumination on our current scenes, especially placed within the context of the main bottleneck of our pipeline, unwrapping (see Section 4.2). Here we can also use the traversal strategy by Fuetterling et al. [13]. In our special case, we always trace a fixed number of final gather rays from a single position on the 3D-model, corresponding to a UV-map pixel. These rays thus share a common origin and can be grouped in packets. As shown for primary rays, packet traversal has the potential to be much faster than single ray traversal.

4.2 Global Illumination Pipeline Performance

To determine whether the global illumination pipeline performance is fast enough for prototyping purposes, we break down the pipeline into steps and measure how long each step takes to complete. Summing their contributions gives us the total time that the global illumination pipeline takes to complete for the tested scenes. Using this, we determine whether the pipeline will be fast enough for larger scenes.

4.2.1 Experiment Setup

In our experiment, we break down the pipeline into five steps. For each of the five steps, we measure how long it takes to complete using a stopwatch in code. Their contribution is summed to determine the total pipeline duration. The parameter values governing the steps in the pipeline are given in Table 4.4. The following five steps are discerned within the pipeline:

1. *Unwrapping* - Invoking UVAtlas on parts of the model to determine UV maps (Section 3.2.3).
2. *Photon Map Construction* - The tracing of photons throughout the scene and constructing the uniform grid to store them (Section 3.2.2).
3. *Radiance Baking* - The baking of radiance maps according to Section 3.2.4.
4. *FG Baking* - The baking of indirect illumination maps using final gather as described in Section 3.2.5.
5. *Light Map Composition* - The combining of deterministic direct illumination with the indirect illumination maps produced using final gather (Section 3.2.6).

For each of these steps, we measure the time they take to complete. This is done for three consecutive runs, after which the resulting measurements are averaged. Our results can be seen in Section 4.2.2.

Parameter	Sponza	San Miguel	Powerplant
BVH Bin Count	64	64	64
BVH Max Prims per Leaf	3	3	3
Photon Count	1.000.000	1.000.000	2.000.000
Photon Max Bounces	5	5	5
Final Gather Ray Count	32	32	10
Blur Kernel Size	3	3	2
Radiance Map Resolution	128x128	96x96	Adaptive, 6-76
Final Gather Map Resolution	256x256	128x128	Adaptive, 32-304
Light Map Resolution	1024x1024	512x512	1024x1024
Number of Maps (parts for unwrap)	8	197	355
Number of Threads	12	12	12

TABLE 4.4: Parameter values governing the steps in the pipeline for each scene.

4.2.2 Results

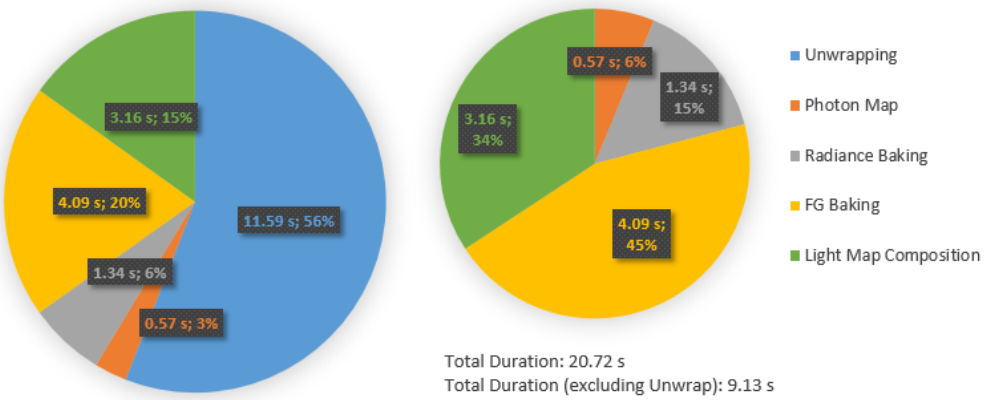

FIGURE 4.1: A breakdown of the duration of steps in the global illumination pipeline for the *Sponza* scene. For each section of the chart, the duration is indicated in seconds, along with what percentage of the pipeline it makes up. The left pie-chart shows the total pipeline, the right pie-chart omits the unwrapping step.

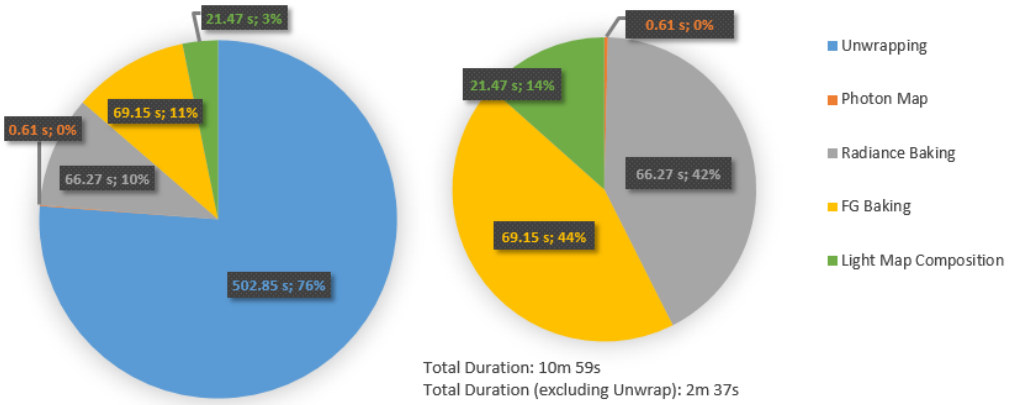

FIGURE 4.2: A breakdown of the duration of steps in the global illumination pipeline for the *San Miguel* scene. For each section of the chart, the duration is indicated in seconds, along with what percentage of the pipeline it makes up. The left pie-chart shows the total pipeline, the right pie-chart omits the unwrapping step.

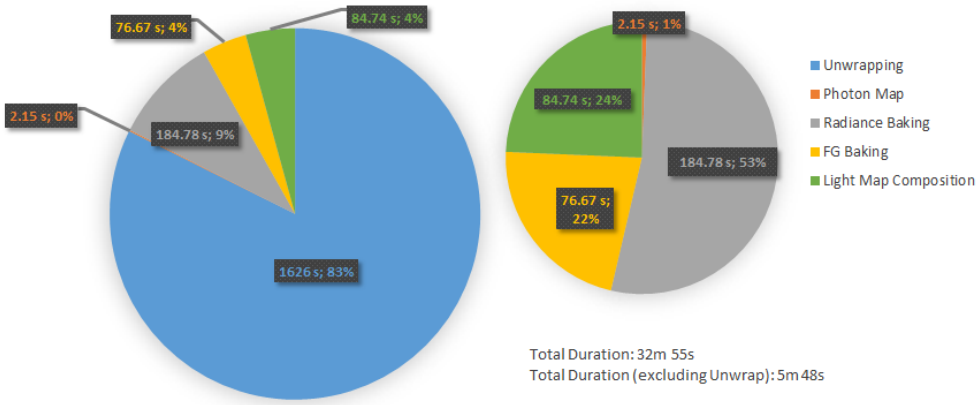

FIGURE 4.3: A breakdown of the duration of steps in the global illumination pipeline for the *Powerplant* scene. For each section of the chart, the duration is indicated in seconds, along with what percentage of the pipeline it makes up. The left pie-chart shows the total pipeline, the right pie-chart omits the unwrapping step.

4.2.3 Discussion

All three breakdowns of the duration of the steps in the pipeline show that unwrapping clearly takes up the most time. This is especially visible for the Powerplant scene, where UV map generation takes up over 80% of the total processing time. Unwrapping is outside of the scope of this thesis and handled by an external tool. Attempts have been made to mitigate the effects of the slow unwrapping process, by splitting up the model into parts.

Disregarding the unwrapping time, we can analyze the rest of the pipeline. Each figure also shows a breakdown for the pipeline without the unwrapping step. Here, we can clearly see that photon map construction is the shortest step in the pipeline. Radiance map baking and FG map baking are either the largest, or second largest

contributors to the total time. We note that all our tested scenes are processed within the hour, even if unwrapping is included. If unwrapping is omitted, our larger scenes are processed within several minutes. Clever solutions that only unwrap modified parts of the scene can be thought of, to avoid having to unwrap the entire model at every application start-up.

Radiance baking time increases when the number of photons in the photon map is increased. This can clearly be observed for the Powerplant scene, where the number of photons is doubled compared to other scenes. However, higher photon counts are required to illuminate all regions in a large scene. Radiance baking time may be lowered by improving the photon grid datastructure search process.

Final gather time is affected quite negatively by increasing the number of triangles in the scene. Furthermore, when the number of triangles increases, our FG map resolution must also be higher, because more detail will be present. This is mitigated by our incremental updates, as discussed in Section 3.3. Fortunately, the number of FG rays can be lowered for larger models, allowing detail to accumulate at runtime. Final gather time can be decreased by improving BVH traversal, as discussed in Section 4.1.

Light map writing also takes up a significant portion of the total duration of the pipeline. The light map resolution must be high, to preserve crips shadows. This step may be avoidable in the future, if only the indirect light maps are exported and combined with shadow mapping and local shading. This also proves to be beneficial for image quality, as shown in section 4.3.

If we assume that scaling of our total pipeline duration is linear and we use the breakdown for the Powerplant as a starting point for a linear extrapolation, we can conclude that a scene containing one billion triangles will take our pipeline approximately eight hours to process. However, when scenes contain more triangles, the total area of the scene does not neccesarily increase. The area of the scene is directly correlated to the desired resolution of the lowest light map LOD and therefore, it may not be required to increase the resolution of the light maps linearly. Furthermore, we can lower the resolution of the lowest light map LOD and increase the number of LODs, thereby moving computation time from start-up to runtime.

4.3 Visual Quality

In this section, we attempt to determine whether the global illumination pipeline produces light maps that are sufficiently accurate for the prototyping purposes described in Chapter 1. To this end, a path tracer (Section 2.3.2) has been implemented in the global illumination framework that is capable of rendering photorealistic images. Comparisons will be made between the pathtracer and two of our approaches.

4.3.1 Experiment Setup

To evaluate the quality of the produced lighting, we take two camera viewpoints within the Sponza scene, one within the San Miguel scene and one within the Powerplant scene to confirm that our pipeline also produces correct results as scenes grow larger. At each viewpoint, we produce three images. The left image in our results is rendered using our photorealistic path tracer. The second is rendered using ray tracing. In this middle image, indirect illumination is determined using the final gather map and direct illumination is determined using shadow rays and local shading. The third image, on the left, is rendered using ray tracing, but uses primary rays

only to directly visualize the composite light map, which includes shadows, direct illumination and indirect illumination.

All parameters for these experiments are the same as those use for the global illumination pipeline experiments in Section 4.2, shown in Table 4.4.

4.3.2 Results

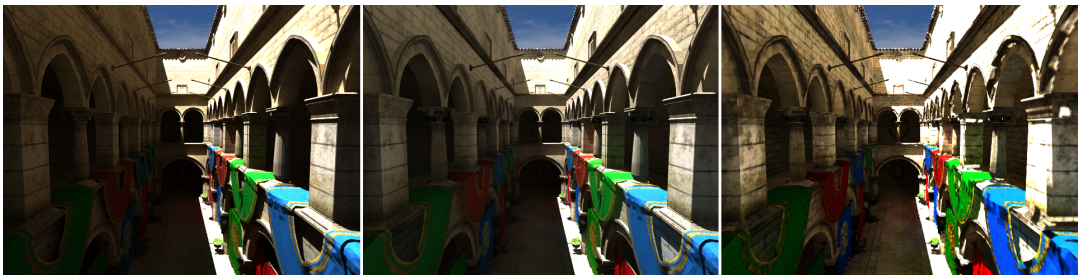

FIGURE 4.4: A pathtraced render of Sponza using one point light as a sun (left), compared to our approach using indirect illumination maps (middle) and our composite light map approach (right).

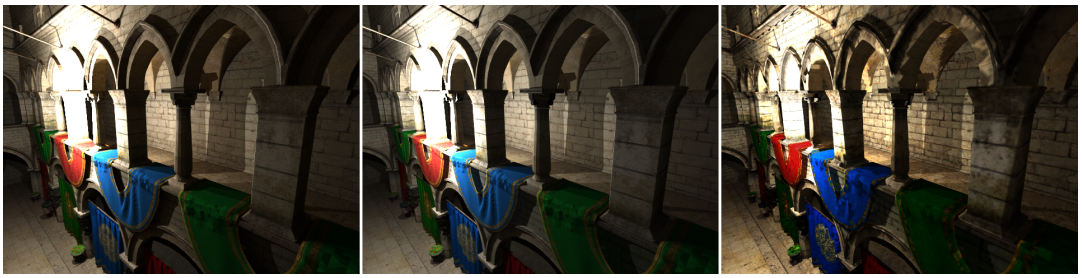

FIGURE 4.5: A pathtraced render of Sponza using two point lights (left), compared to our approach using indirect illumination maps (middle) and our composite light map approach (right).

FIGURE 4.6: A pathtraced render of San Miguel using one point light (left), compared to our approach using indirect illumination maps (middle) and our composite light map approach (right).

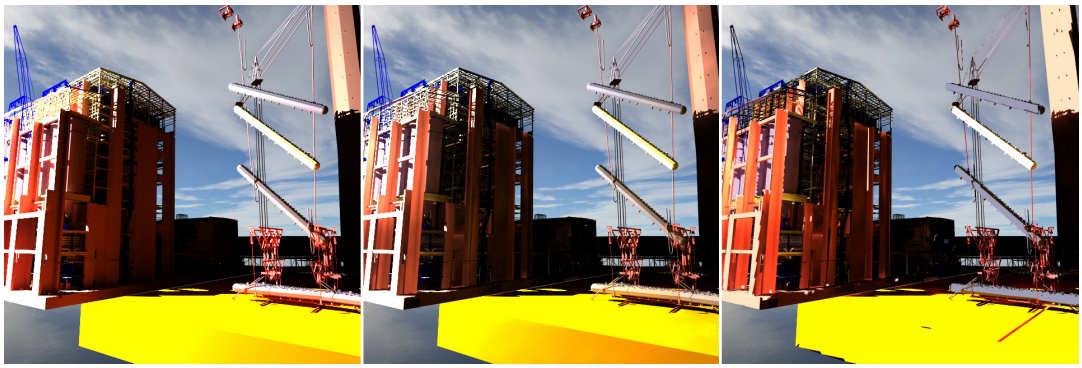

FIGURE 4.7: A pathtraced render of Powerplant using one point light (left), compared to our approach using indirect illumination maps (middle) and our composite light map approach (right).

4.3.3 Discussion

Our approach, shown in the middle and on the right of Figures 4.4, 4.5, 4.6 and 4.7 differs from the photorealistic image produced by the pathtracer. However, our approach is not intended to be completely photorealistic, as discussed in Chapter 1. If lighting quality is sufficiently accurate, it can be used for prototyping and previewing purposes. Our approach for rendering the middle image produces lighting quality that is sometimes hard to distinguish from a path traced render, especially without the direct comparison. This approach thus seems to produce the most accurate results with a relatively short startup time. The image quality on the right is most easily transfered to a rasterization renderer, as these are simply textures that can be exported from our pipeline directly. However, the effect that is created in the middle image can also be transferred to a rasterization renderer, by implementing techniques such as shadow mapping and local shading.

We conclude that the quality of the produced lighting is sufficiently accurate for our purposes, taking into account the short processing times of our global illumination pipeline. On larger scenes, our final composite light map may show artefacts for more complicated scenes. These stem from inadequate performance of the external unwrapping tool. By using adaptive light map sizes, in which the size of the light map depends on the actual area of the triangles it covers, many artefacts are removed. This is visualized by the increase in lighting quality between the San Miguel scene and the Powerplant scene. At the moment, it is recommended to use our indirect illumination maps in combination with the shadow maps in a rasterizer or, in combination with shadow rays, in a ray tracer.

Chapter 5

Conclusion

In this master's thesis, we first introduced our problem of precomputing global illumination for massive scenes in Chapter 1. We formulate three research questions which, when answered should help solve rendering these massive scenes with global illumination. Then, we review relevant literature to our approach in Chapter 2. We use these preliminaries as building blocks for our global illumination pipeline, presented in Chapter 3. Our pipeline first constructs a photon map and uses an external tool to generate one or more 2D parametrizations of the surface of the scene. These parametrizations are used to generate radiance maps and final gather maps, which contain the indirect lighting in the scene. The indirect lighting is then either composited at runtime with direct lighting in a ray tracer, or combined with direct lighting and baked to a single texture. We also present two methods for updating the final gather maps, increasing the quality of rendered lighting at run-time.

Our results are then evaluated in three parts in Chapter 4. First, we evaluate our implementations of algorithms that concern the BVH and MBVH in Section 4.1. Both construction speeds and traversal speeds are evaluated. Our findings indicate that large performance gains may be achievable, but are not required at this point for testing and development purposes. Then, the duration of individual steps in the global illumination pipeline is measured in Section 4.2. The key bottleneck of our pipeline is UV-mapping, which is done using an external tool. Disregarding UV-mapping, we conclude that baking the radiance map and the final gather map are key contributors to the global illumination pipeline running time. In Section 4.3, we evaluate the visual results of pipeline, concluding that live composition of direct and indirect lighting is our best result.

Using our results, we can now answer our research questions.

1. **How can global illumination for scenes consisting of billions of triangles be precomputed fast enough for prototyping purposes described in Chapter 1?** - By implementing a global illumination pipeline that precomputes our information using the steps outlined in Section 3.2, coarse global illumination can be precomputed rapidly. Extrapolation of our results indicates that global illumination for scenes consisting of a billion triangles can be precomputed within eight hours using our system, disregarding UV-mapping and BVH construction time. However, we have implemented a system that updates light maps at run-time, allowing us to lower the initial resolution of the light maps and speed up precomputation. In this manner, lighting becomes increasingly accurate at run-time. We have determined that the resulting lighting is sufficiently accurate enough for our prototyping purposes.

2. **Is photon mapping a suitable method for fast precomputation of global illumination on massive scenes?** - Yes, photon mapping is suitable because photon map construction time is extremely low compared to the other steps in

the pipeline. We show that photon maps can be constructed extremely rapidly, even for massive scenes. The steps required for transferring the information stored in the photon map to a format that is suitable for fast rendering takes up the bulk of the time in our global illumination pipeline. However, these steps can be sped up by lowering the initial light map resolution, thereby moving computation time from start-up to runtime.

3. **Does the theoretical logarithmic complexity of BVH ray traversal hold up for very large scenes?** - It does not. This research question is answered in more detail in Appendix A.

5.1 Future Work

The global illumination pipeline is by no means complete. Many topics have been touched upon in our related work chapter, that are not included in the pipeline. For example, the pipeline can be further extended by supporting specular and translucent materials. Implementing advanced material properties, such as those described in Section 2.3.4, is another potential direction for future work. Furthermore, area lights and soft shadows currently only have a very rudimentary implementation that places tremendous strain on the pipeline running time.

As discussed in Chapter 4, much work can be done to speed up both BVH construction and traversal. By implementing algorithms such as those developed by Fuetterling et al. [13], significant performance gains should be achievable. Therefore, we suggest that future work focuses on improving the quality of the illumination by adding the missing elements described above, while improving the acceleration structures that they depend upon. Special emphasis must remain to be placed on keeping the total execution time of the pipeline as low as possible.

5.2 Acknowledgements

I would like to thank my supervisor for his continued support and input considering this research project. His help with the theoretical background, practical implementation and constuctive criticism for the thesis was invaluable. Furthermore, I am grateful for the support by Stirling Labs Ltd, providing me with a framework for my research and hardware to carry out my experiments. I also thank the authors of the models I have used for testing whice have been downloaded from McGuire's Computer Graphics Archive [28].

Appendix A

Scalability of Ray Traversal Time for the BVH in Practice

Scalability of Ray Traversal Time for the BVH in Practice

Casper Schouls

September 26, 2017

Abstract

In this paper, we investigate the time complexity of ray traversal. Theoretically, ray traversal time complexity scales logarithmically. In practice, the behaviour of the memory hierarchy affects the performance of ray tracing renderers. We show how rendering time is affected by gradually increasing the number of triangles in a standardized scene for both single ray traversal and ranged traversal using packets. Our results indicate that the assumed logarithmic scaling of BVH traversal time underestimates real-world BVH traversal performance, for both single ray traversal as well as ranged traversal. Furthermore, the coherence of the ray set does not appear to affect the scaling of rendering time. Extrapolation of our results indicates that single ray traversal may become faster than ray packet traversal as scenes contain more than a hundred billion triangles.

1 Introduction

Rendering using ray tracing has the potential to scale logarithmically in the number of rendered primitives using acceleration structures such as the *bounding volume hierarchy* (BVH). Once such a BVH has been constructed for the given set of primitives, ray traversal time can scale logarithmically in the number of rendered primitives. This scaling behaviour is much better than that of traditional forward renderers, where rendering time typically scales linearly in the number of rendered primitives. Therefore, ray tracing has the potential to render large scenes significantly faster than forward renderers.

While theoretically ray traversal time through a BVH scales logarithmically, in practice, many other factors affect the performance of a ray tracing renderer. When rays are highly coherent, which is the case for primary rays, they tend to intersect the same geometry. This improves CPU cache performance. Incoherent ray sets, such as extension rays, have a lower probability of intersecting the same geometry and traversing the same nodes of the BVH.

Furthermore, when geometry becomes small, the likelihood of two rays intersecting the same geometry decreases. This incurs higher cache miss rates and more lookups from main memory. Packet traversal, as described by Overbeck et al.[2], attempts to reduce this effect by amortizing memory lookup cost over multiple rays.

In this paper, we investigate how traversal time of primary rays scales using both ordered single ray BVH traversal and an implementation of ranged traversal using ray packets. Furthermore, we investigate how diminishing coherence of ray sets affects traversal time and scaling. Our research questions are as follows:

1. Does the theoretical logarithmic complexity of BVH ray traversal hold up in practice?
2. What is the effect of the ray traversal scheme on scaling of BVH traversal time?
3. How does diminishing coherence of ray sets affect scaling of BVH traversal time?

Section 2 describes our research methodology. The results of our measurements are described and discussed in section 3. Lastly, we answer our research questions and predict how ray traversal time will scale to even larger scenes in section 4.

2 Research Methods

We set up four types of experiments to measure the effects of several parameters on ray traversal time. For each experiment, we measure the duration of tracing a precomputed set of rays through a scene. This is done for 256 iterations and the average render time over these iterations is taken.

Our scene setup is a Menger sponge of level 1, rendered from the perspective shown in figure 1. The Menger sponge allows us to construct a completely balanced BVH and thus negates effects of inbalanced trees on render time. By subdividing the triangles, we increase the number of triangles in the scene while maintaining the same tree layout. Using our subdivision scheme, each triangle is split into four subtriangles. We use midpoint splits to build our BVH and stop when a node contains less than four triangles.

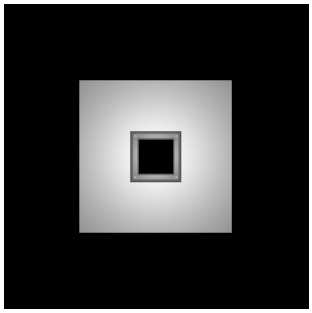

Figure 1: The viewpoint from which we generate primary rays.

All rays are generated before starting our experiment and are stored in main memory. During the experiment, the stored rays traverse the BVH. No shading is applied afterwards. Our implementation is single-threaded and does not utilize SIMD instructions. The ray batches which we generate are always of size 409,600 (640x640), both for primary and random rays. All primary rays are generated from the camera position shown in figure 1.

2.1 Experiment 1 - Single Ray Traversal, Primary Rays

In our first experiment, we measure scaling of render time for ordered single ray traversal of primary rays. We generate a set of 640x640 rays for a virtual screen plane, with one ray per pixel.

2.2 Experiment 2 - Ray Packet Traversal, Primary Rays

In our second experiment, we measure scaling of render time for ray packet traversal of primary rays. We generate a set of ray packets for a virtual screen plane, spanning 640x640 pixels, with one ray per pixel. Our packet size varies, ranging from 2x2 to 32x32. The packet traversal scheme we use is described by Overbeck et al. [2] as *ranged traversal*.

2.3 Experiment 3 - Ray Packet Traversal, Rays of Diminishing Coherence

Our third experiment is intended to bridge the gap between highly coherent primary rays (section 2.2) and highly incoherent random rays (section 2.4). We generate rays in packets as in section 2.2, but increase the distance between ray targets on the virtual screen plane gradually. This effectively increases the angle between the rays in the packet, as done previously in J. Bikker's PhD thesis[1]. This diminishes coherence in the ray set because rays are now more divergent and less likely to intersect the same triangles. We investigate how render time depends on both coherence of the rays in the ray packet and the number of triangles in the scene.

Like our experiment for ray packets, described in section 2.2, we generate rays for a virtual screen plane spanning 640x640 pixels. The distance between ray targets is now no longer fixed at one, but can vary. This causes packets to overlap and diminishes coherence in the rays in the packet. Our packet size is fixed at 4x4 and the distance between ray targets ranges from one to ten. Our packets traverse the BVH using ranged traversal[2].

2.4 Experiment 4 - Single Ray Traversal, Random Rays

In our final experiment, we measure scaling of render time for ray packet traversal of random rays. These random rays are generated by picking a starting point on the bounding sphere containing the Menger sponge. A random ray target is selected on this bounding sphere and is used to generate the ray direction. The same number of rays is traced as in the other experiments, except now they are no longer generated by the camera. The ray traversal scheme we use is basic ordered single ray traversal.

3 Results and evaluation

In this section, we present the results for the experiments described in section 2. Each of these results is accompanied by a short discussion.

3.1 Experiment 1 - Single Ray Traversal, Primary Rays

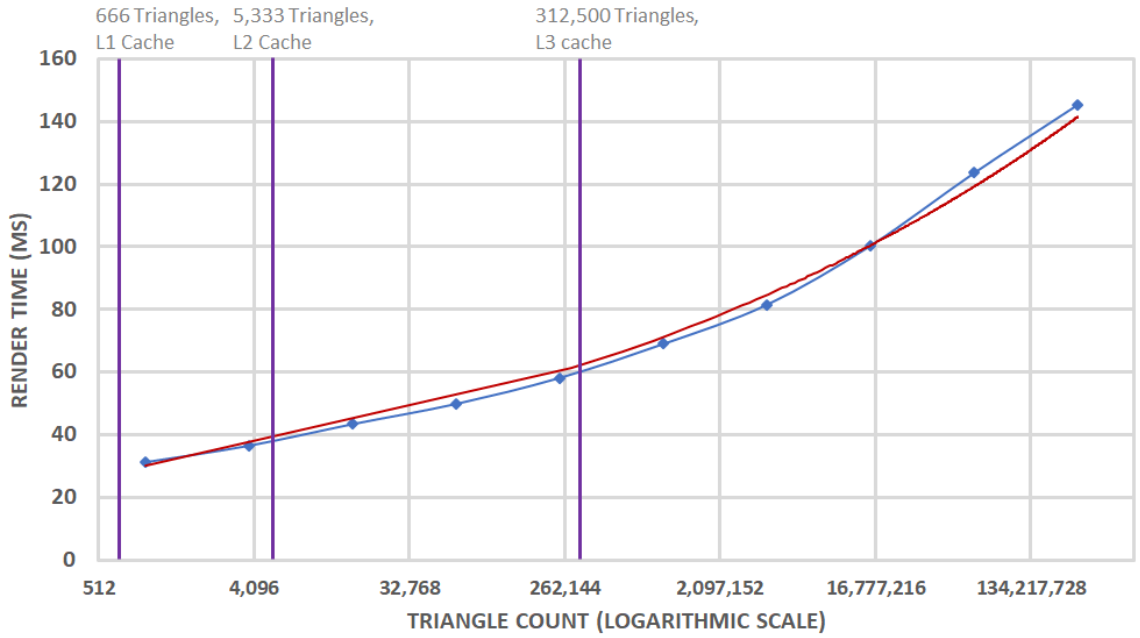

Figure 2: Rendering times for single ray traversal (y) as the triangle count increases logarithmically (x).

Our results for our first experiment (2.1), shown in figure 2, show the average render time in milliseconds on the y-axis and the triangle count for that scene on the x-axis as the blue line. Our x-axis uses a logarithmic scale of base two, which means that theoretical predictions of the render time should follow a straight line. As we can see in figure 2, this is clearly not the case. The purple lines indicate when a scene no longer fits within the mentioned CPU cache.

Render times at first do appear to grow logarithmically: between triangle counts of 980 through 245,760, the graph shows a fairly straight line. The same can be seen between triangle counts of 983,040 and 251,658,240. There thus appears to be a break between these where render times start increasing faster. At this point, the L3 cache of our CPU seems to overflow, causing render times to increase faster. In our case, a more accurate approximation of single ray render times can be given using equation 1. This equation is plotted in figure 2 in red.

$$y = 13x^{\frac{1}{8}} \tag{1}$$

with (for all equations in this paper)

- y is the average render time in milliseconds.
- x is the number of triangles in the scene.

Using this equation, we predict that rendering a single frame of the Menger sponge, consisting of a billion triangles, at a resolution of 640x640 will take approximately 173 ms. Increasing the number of triangles to one hundred billion would then allow a single frame to render in approximately 308 ms, not

even doubling the render time. Our solution is relatively unoptimized and does not use SIMD instructions to speed up processing. Therefore, real-world render times of an optimized implementation might be even lower. Furthermore, our implementation is single-threaded and as ray tracing is easy to parallelize, performance could be increased greatly.

3.2 Experiment 2 - Ray Packet Traversal, Primary Rays

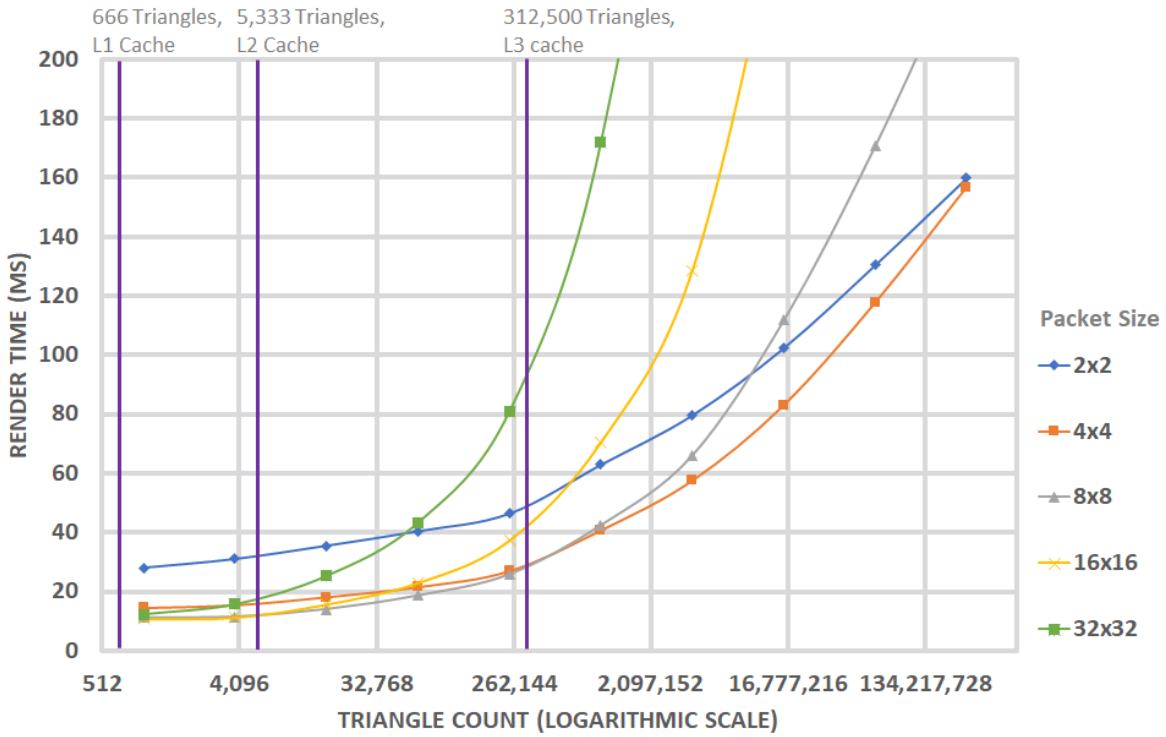

Figure 3: Rendering times in ms for ranged traversal using ray packets (y-axis) as the triangle count increases logarithmically (x-axis) for primary rays. Packet sizes ranging from 2x2 to 32x32 are displayed.

The results for our second experiment (section 2.2) are shown in figure 3. Each line in the graph represents a different packet size. The x-axis uses a logarithmic scale of base two for the triangle count and the y-axis shows the average render time in milliseconds. These plots are similar to the graph for single ray traversal (figure 2). The purple lines again indicate when a scene exceeds the bounds of our CPU caches.

Average render time using ray packets clearly does not scale logarithmically. Here, we observe the same break in the line as with single ray traversal, especially visible for the smaller ray packets. Enlarging the packet improves performance at low triangle counts, but smaller ray packets outperform large packets for higher triangle counts. For the tested triangle counts, 4x4 packets appear to be ideal: they accomodate adequate traversal speeds at low triangle counts, while maintaining high performance as the triangle count increases. However, extrapolation would seem to indicate that 4x4 packets will be outperformed by 2x2 packets at some point. Therefore, small packet sizes appear to be more suitable for scenes containing billions of triangles, perhaps even switching to single ray traversal once this becomes faster.

Ray packet render time can be approximated using equations 2 and 3 for 2x2 and 4x4 packets. See equation 1 for variable definitions.

$$y = 8.9x^{\frac{1}{7}} \tag{2}$$

$$y = 2.7x^{\frac{1}{5}} \tag{3}$$

Evaluating the 2x2 equation for one bilion, and one hundred billion triangles gives us a expected render times of 171 ms and 331 ms. The 4x4 variant results in render times of 170 ms and 427 ms, respectively. According to our measurements and approximations, single ray traversal will thus be the fastest option for one hundred billion triangles (see section 3.1). We do note that our ray packet traversal implementation is straightforward and does not utilize SIMD to test multiple rays concurrently against multiple bounding boxes or triangles. Doing so might increase performance so much that the turning point for switching to single ray traversal may occur at much higher triangle counts.

3.3 Experiment 3 - Ray Packet Traversal, Rays of Diminishing Coherence

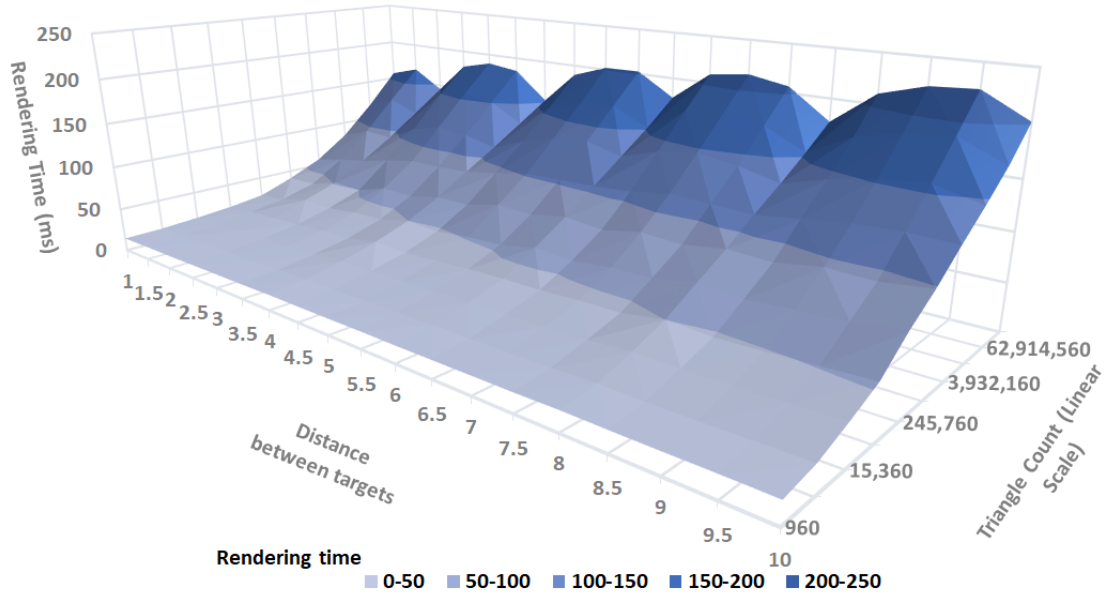

Figure 4: A three-dimensional plot showing the effects of the angle between rays (horizontal axis), the triangle count (depth-axis) on the rendering time (in ms, vertical axis) using ranged traversal.

The results of our experiments using rays of diminishing coherence (section 2.3) can be seen in figure 4. This three dimensional plot shows the distance between ray targets on the horizontal axis, the number of triangles on the depth axis and the resulting render times (in milliseconds) on the vertical axis. Here, the depth axis is not logarithmic.

The graph shows that render times increase as the distance between ray targets increases. We can see that the coherence of a ray set has few effects on the scaling characteristics of ray tracing: incoherent rays respond to an increasing number of triangles in the same manner as coherent rays. The only difference is that render times are consistently higher for incoherent rays.

We also note that distances between ray targets that are multiples of two appear to run faster than those that are not.

3.4 Experiment 4 - Single Ray Traversal, Random Rays

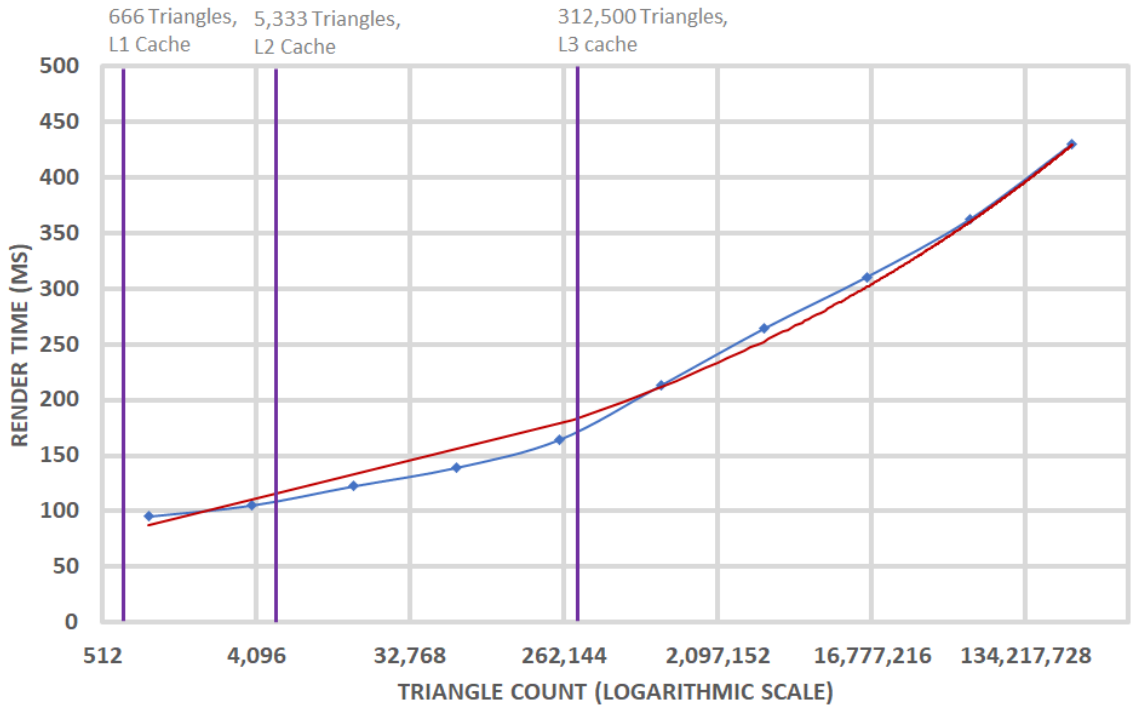

Figure 5: Rendering times for single ray traversal (y) of random rays as the triangle count increases logarithmically (x).

Figure 5 shows the results for our experiments for randomly generated rays, as described in section 2.4. This graph is similar to those for single rays and ray packets: the x-axis is base two logarithmic and shows the triangle count, while the y-axis shows the average render time in milliseconds. The blue line are the results of our measurements, while the red line is the approximation given by equation 4. Once again, the line is not straight and thus does not represent logarithmic scaling. The purple lines indicate when a scene exceeds the bounds of our CPU caches.

Our results for single ray traversal of primary rays indicate a break in the line between 245,760 and 983,040 triangles. The same break can be observed here, but it occurs earlier, between 61,440 and 245,760 triangles. If we assume that this is indeed caused by overflooding caches, we can speculate that CPU cache policies are less effective at predicting which data will be required next for random rays than they are for primary rays. This might cause the break to occur earlier.

Equation 4 approximates the average render time for random rays. Using this equation, we predict that rendering times for one billion and one hundred billion triangles are at 488 ms and 868 ms. This is significantly longer compared to the approximated render times for primary rays (section 3.1), showcasing the impact of coherence of ray sets on rendering time. See equation 1 for variable definitions.

$$y = 36.6x^{\frac{1}{8}} \tag{4}$$

4 Conclusion and Future Work

In this paper, we have described four different experiments for testing the effects of increasing the number of triangles on the rendering time of a ray tracer. We have shown how logarithmic scaling of rendering time is an underestimate of real-world scaling. Using our results, we can answer our research questions clearly and concisely:

1. **Does the theoretical logarithmic complexity of BVH ray traversal hold up in practice?** - It does not: logarithmic scaling is an underestimate of real-world BVH ray traversal performance. Furthermore, our experiments show that a triangle count exists, which when exceeded causes render times to grow faster. This triangle count likely depends on the size of the CPU caches.
2. **What is the effect of the ray traversal scheme on scaling of BVH traversal time?** - The ray traversal scheme affects scaling significantly. Small packets scale better than larger packets, while larger packets provide better performance at low triangle counts. Large packets scale less well than smaller packets, but provide better performance at low triangle counts. As triangle counts get larger (in our case, around a hundred billion), single ray traversal begins to outperform ray packet traversal.
3. **How does diminishing coherence of ray sets affect scaling of BVH traversal time?** - Diminishing coherence of ray sets does not affect scaling of BVH traversal time. It does affect rendering time, but the manner in which these scale is the same, regardless of the coherence of the ray set.

Future work may focus on validating our assumption that the break in scaling is caused by the CPU caching policy. A subdivision scheme or a scene that allows for a more gradual increase in the number of triangles, while maintaining the same BVH topology, can confirm that this break is indeed present.

References

[1] Jacco Bikker. "Ray tracing in real-time games". PhD thesis. Delft University, 2012.

[2] Ryan Overbeck, Ravi Ramamoorthi, and William R Mark. "Large ray packets for real-time whitted ray tracing". In: *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on*. IEEE. 2008, pp. 41–48.

Bibliography

[1] . *Stirling Labs Ltd.* 2017. URL: http://www.stirlinglabs.com (visited on 09/05/2017).

[2] John Amanatides and Andrew Woo. "A Fast Voxel Traversal Algorithm for Ray Tracing". In: *In Eurographics '87*. 1987, pp. 3–10.

[3] Rasmus Barringer and Tomas Akenine-Möller. "Dynamic Ray Stream Traversal". In: *ACM Trans. Graph.* 33.4 (July 2014), 151:1–151:9. ISSN: 0730-0301.

[4] Keshav Channa. *Light Mapping - Theory and Implementation*. 2003. URL: http://www.flipcode.com/archives/Light_Mapping_Theory_and_Implementation.shtml (visited on 03/17/2017).

[5] Per H Christensen. "Global illumination and all that". In: *SIGGRAPH 2003 course notes* 9 (2003), pp. 31–72.

[6] Robert L. Cook, Thomas Porter, and Loren Carpenter. "Distributed Ray Tracing". In: *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '84. New York, NY, USA: ACM, 1984, pp. 137–145. ISBN: 0-89791-138-5.

[7] HTC Corporation. *HTC Vive*. 2017. URL: http://www.vive.com (visited on 03/14/2017).

[8] Cyril Crassin et al. "Interactive indirect illumination using voxel cone tracing". In: *Computer Graphics Forum*. Vol. 30. 7. Wiley Online Library. 2011, pp. 1921–1930.

[9] Holger Dammertz, Johannes Hanika, and Alexander Keller. "Shallow bounding volume hierarchies for fast SIMD ray tracing of incoherent rays". In: *Computer Graphics Forum*. Vol. 27. 4. Wiley Online Library. 2008, pp. 1225–1233.

[10] B. Fabianowski and J. Dingliana. "Interactive Global Photon Mapping". In: *Proceedings of the Twentieth Eurographics Conference on Rendering*. EGSR'09. Girona, Spain: Eurographics Association, 2009, pp. 1151–1159.

[11] Fairlight. *Five Faces*. 2013. URL: http://www.pouet.net/prod.php?which=61211 (visited on 03/14/2017).

[12] Fairlight. *Five Faces Brickmap Datastructure*. 2013. URL: https://directtovideo.wordpress.com/2013/05/08/real-time-ray-tracing-part-2/ (visited on 03/14/2017).

[13] Valentin Fuetterling et al. "Efficient Ray Tracing Kernels for Modern CPU Architectures". In: *Journal of Computer Graphics Techniques (JCGT)* 4.5 (2015), pp. 90–111. ISSN: 2331-7418.

[14] Valentin Fuetterling et al. "Parallel spatial splits in bounding volume hierarchies". In: *Proceedings of the 16th Eurographics Symposium on Parallel Graphics and Visualization*. Eurographics Association. 2016, pp. 21–30.

[15] P Ganestam et al. "Bonsai: rapid bounding volume hierarchy generation using mini trees". In: *Journal of Computer Graphics Techniques Vol* 4.3 (2015).

[16] J. Gunther et al. "Realtime Ray Tracing on GPU with BVH-based Packet Traversal". In: *2007 IEEE Symposium on Interactive Ray Tracing*. 2007, pp. 113–118.

[17] Toshiya Hachisuka, Shinji Ogaki, and Henrik Wann Jensen. "Progressive photon mapping". In: *ACM Transactions on Graphics (TOG)* 27.5 (2008), p. 130.

[18] Vlastimil Havran. "Heuristic ray shooting algorithms". PhD thesis. Citeseer, 2000.

[19] Bas Hoogeboom et al. *PathCraft*. 2016. URL: https://www.pathcraft.nl/ (visited on 03/14/2017).

[20] Luxion Inc. *KeyShot*. 2017. URL: http://www.keyshot.com (visited on 03/14/2017).

[21] Frederik W. Jansen. "Data structures for ray tracing". In: *Data Structures for Raster Graphics: Proceedings of a Workshop held at Steensel, The Netherlands, June 24–28, 1985*. Ed. by Laurens R. A. Kessener, Frans J. Peters, and Marloes L. P. van Lierop. Berlin, Heidelberg: Springer Berlin Heidelberg, 1986, pp. 57–73. ISBN: 978-3-642-71071-1.

[22] Henrik Wann Jensen. "Global illumination using photon maps". In: *Rendering Techniques' 96*. Springer, 1996, pp. 21–30.

[23] Henrik Wann Jensen et al. "A practical guide to global illumination using photon mapping". In: ().

[24] James T. Kajiya. "The Rendering Equation". In: *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '86. New York, NY, USA: ACM, 1986, pp. 143–150. ISBN: 0-89791-196-2.

[25] Eric P. Lafortune and Yves D. Willems. "Bi-directional path tracing". In: *Proceedings of Third International Conference on Computational Graphics and Visualization Techniques (Compugraphics '93)*. Alvor, Portugal, 1993, pp. 145–153.

[26] Oculus VR LLC. *Oculus Rift*. 2017. URL: http://www.oculus.com (visited on 03/14/2017).

[27] J David MacDonald and Kellogg S Booth. "Heuristics for ray tracing using space subdivision". In: *The Visual Computer* 6.3 (1990), pp. 153–166.

[28] Morgan McGuire. *Computer Graphics Archive*. https://casual-effects.com/data. 2017. URL: https://casual-effects.com/data.

[29] Morgan McGuire and David Luebke. "Hardware-accelerated Global Illumination by Image Space Photon Mapping". In: *Proceedings of the Conference on High Performance Graphics 2009*. HPG '09. New Orleans, Louisiana: ACM, 2009, pp. 77–89. ISBN: 978-1-60558-603-8.

[30] Ryan Overbeck, Ravi Ramamoorthi, and William R Mark. "Large ray packets for real-time whitted ray tracing". In: *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on*. IEEE. 2008, pp. 41–48.

[31] Bui Tuong Phong. "Illumination for Computer Generated Pictures". In: *Commun. ACM* 18.6 (June 1975), pp. 311–317. ISSN: 0001-0782.

[32] Steven M. Rubin and Turner Whitted. "A 3-dimensional Representation for Fast Rendering of Complex Scenes". In: *SIGGRAPH Comput. Graph.* 14.3 (July 1980), pp. 110–116. ISSN: 0097-8930.

[33] Martin Stich, Heiko Friedrich, and Andreas Dietrich. "Spatial splits in bounding volume hierarchies". In: *Proceedings of the Conference on High Performance Graphics 2009*. ACM. 2009, pp. 7–13.

[34] Niels Thrane, Lars Ole Simonsen, and Advisor Peter Ørbæk. *A comparison of acceleration structures for GPU assisted ray tracing*. Tech. rep. 2005.

[35] K. E. Torrance and E. M. Sparrow. "Theory for Off-Specular Reflection From Roughened Surfaces$*$". In: *J. Opt. Soc. Am.* 57.9 (1967), pp. 1105–1114.

[36] Eric Veach. "Robust monte carlo methods for light transport simulation". PhD thesis. Stanford University, 1997.

[37] I. Wald. "On fast Construction of SAH-based Bounding Volume Hierarchies". In: *2007 IEEE Symposium on Interactive Ray Tracing*. 2007, pp. 33–40.

[38] I. Wald and V. Havran. "On building fast kd-Trees for Ray Tracing, and on doing that in O(N log N)". In: *2006 IEEE Symposium on Interactive Ray Tracing*. 2006, pp. 61–69.

[39] Ingo Wald. "On fast construction of SAH-based bounding volume hierarchies". In: *Interactive Ray Tracing, 2007. RT'07. IEEE Symposium on*. IEEE. 2007, pp. 33–40.

[40] Ingo Wald, Carsten Benthin, and Solomon Boulos. "Getting rid of packets-efficient simd single-ray traversal using multi-branching bvhs". In: *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on*. IEEE. 2008, pp. 49–57.

[41] Ingo Wald et al. "Interactive Rendering with Coherent Ray Tracing". In: *Computer Graphics Forum* 20.3 (2001), pp. 153–165. ISSN: 1467-8659.

[42] Ingo Wald et al. "Ray Tracing Animated Scenes Using Coherent Grid Traversal". In: *ACM SIGGRAPH 2006 Papers*. SIGGRAPH '06. Boston, Massachusetts: ACM, 2006, pp. 485–493. ISBN: 1-59593-364-6.

[43] Turner Whitted. "An Improved Illumination Model for Shaded Display". In: *ACM SIGGRAPH 2005 Courses*. SIGGRAPH '05. Los Angeles, California: ACM, 2005.