Masarykova univerzita
Fakulta informatiky

Bi-directional Path Tracing on GPU

Master Thesis

Vilém Otte

Brno, Spring 2015

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Vilém Otte

Advisor: RNDr. Marek Vinkler Ph.D.

Acknowledgement

Foremost, I wish to express my sincere gratitude to Principal of the Faculty and Dean of the faculty for Principal's program and Dean's grant (projects *MUNI/C/0941/2013* and *MUNI/33/12/2013*), which allowed me to work on the NTrace project, further extend it and demonstrate my results on CESCG conference. This thesis is written based on my work on the NTrace project.

Further, I would like to thank my advisor, RNDr. Marek Vinkler Ph.D. for the continuous support of my work, for his patience, enthusiasm and knowledge.

I take this opportunity to express gratitude to all of the faculty members for their knowledge and support. I also thank my parents for attention, support and encouragement.

Also, I place on record my sense of gratitude to one and all who directly or indirectly supported me in creating of this thesis.

Abstract

Computer graphics renderers for creating photo-realistic images use mainly unidirectional path tracing, having good results for scenes without caustics or hard cases. There are also few renderers with bi-directional path tracing implementation, however due to the complexity of the algorithm implementation, they almost exclusively target sequential CPUs.

The thesis proposes a way of implementation of bi-directional path tracer on a parallel many-core architectures such as the GPU and provides a working implementation. Further this implementation is compared to a parallel implementation of the standard, unidirectional path tracer, in terms of quality and speed.

Interactive frame rates have been achieved for parallel bi-directional path tracing, proving the possibility of using such algorithm on GPUs.

Keywords

Rendering, Sampling, Global Illumination, Monte-Carlo, CUDA, Ray tracing, Path tracing, Bidirectional Path Tracing, Bidirectional Reflectance Distribution Function

Contents

1 Introduction

Global illumination research currently focuses on two major areas: unbiased methods and real time global illumination. Unbiased methods are such methods, that under some assumption converge to a physically based solution.

One of the major unbiased techniques is the path tracing, this technique is used throughout many industries, like the movie industry, as the rendering algorithm of choice. The bi-directional path tracing is an extension of standard, unidirectional path tracing algorithm, with better results for scenes with caustics or hard cases [1].

Bi-directional path tracing implementations are mostly done in a sequential manner on the CPU, although there exist few of the implementations on the GPU. Sadly they just use graphics hardware only for part of the computation, or they are typically limited in some ways (e.g. they do not support texturing, surface shading, allow only for sphere rendering, etc.).

The presented work in the thesis focuses on the proposal of a method to implement parallel bi-directional path tracing algorithm, and to provide working implementation that runs on graphics hardware.

Furthermore, other GPU-based renderers are compared against this new renderer in terms of both, quality and performance.

---

1. Scene with hard case is such scene where most of the visible geometry is lit by the light generated behind a occluder, typical example is room interior with only indirect light passing through window.

2 Theory

The following chapter provides description and derivation of a technique for image synthesis in form of the rendering equation, closely resembling light behavior in the physics. Such equation can be used for the computation and generation of an image from 3-dimensional scene described using geometry, materials (and lights), viewed from given virtual camera.

Further, the theory behind the scene description is provided, along with proposed model for efficient computation of the rendering equation.

2.1 Rendering Equation

The goal of the computation is a synthesis of an image, describing a view of defined scene by specific camera. Such computation requires calculation of how much energy emitted by lights (surfaces that emit light energy in a virtual scene) reaches the camera. This is actually a simulation of light.

For the purpose of simplicity, following rules apply:

- The light moves along perfectly straight lines.
- The light moves at infinite speed.

Therefore several fundamental laws and definitions of radiometry and geometry have to be defined.

2.1.1 Definitions

Radiant Energy

Denoted as: Q_e
Units: J (joule)
Describes the energy of electromagnetic radiation.

Radiant Flux

Denoted as: Φ_e
Units: W (watt)

$$\Phi_e = \frac{\partial Q_e}{\partial t} \tag{2.1}$$

Describes the amount of energy passing through an area in given time.

Irradiance

Denoted as: E_e
Units: $W \cdot m^{-2}$ (watt per square metre)

$$E_e = \frac{\partial \Phi_e}{\partial A} \tag{2.2}$$

Describes the amount of radiant flux arriving per unit of surface area. In case of radiant flux leaving the surface, it is denoted as radiosity (denoted as J_e). The radiant flux emitted by the surface is denoted as radiant exitance (denoted as M_e), the radiant flux is emitted component of radiosity.

Solid Angle

Denoted as: Ω
Units: sr (steradian)

$$\partial\Omega = \frac{\partial A \cdot cos\theta}{r^2} \tag{2.3}$$

The solid angle Ω subtended by the surface A is defined as the surface area of a unit sphere covered by the projection of the surface onto this sphere, see 2.1.

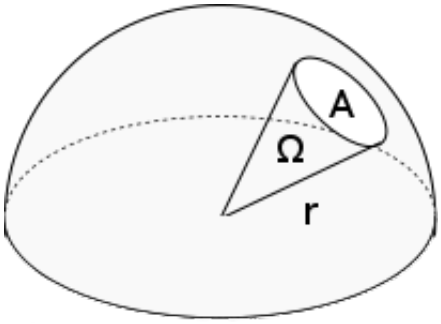

Figure 2.1: This image graphically represents solid angle Ω, on a sphere with radius r, subtending the area A.

Radiant Intensity

Denoted as: $I_{e,\Omega}$
Units: $W \cdot sr^{-1}$ (watt per steradian)

$$I_{e,\Omega} = \frac{\partial\Phi}{\partial\Omega} \tag{2.4}$$

Describes the measure of radiant flux per unit solid angle.

Radiance

Denoted as: $L_{e,\Omega}$
Units: $W \cdot sr^{-1} \cdot m^{-2}$ (watt per steradian per square metre)

$$L_{e,\Omega} = \frac{\partial^2 \Phi_e}{\partial \Omega \cdot \partial A \cdot cos\theta} \tag{2.5}$$

Describes radiant flux emitted, reflected, transmitted or received by a surface per unit solid angle per projected area. It closely resembles color and does not change over distance, such properties make radiance ideal for further computation.

2.1.2 Fundamental Law of Photometry

The following texts provides basics for the derivation and the actual derivation of the rendering equation, similar to Kajiya [6].

Before the actual derivation of the rendering equation, the fundamental law of photometry has to be defined. For the following step, let us assume the simpliest case:

Let there be two elements, denoted as A and A'. The distance between these two elements is equal to r. Therefore radiant flux leaving the first element and reaching the second element - e.g. how much of the emitted energy by one surface reaches the other surface is:

$$\partial \Phi = L_{e,\Omega} \cdot \partial A \cdot cos\theta \cdot \partial \Omega' \tag{2.6}$$

Solid angle in this equation can be expressed by projected area of the element, therefore:

$$\partial \Phi = \frac{L_{e,\Omega} \cdot \partial A \cdot cos\theta \cdot \partial A' \cdot cos\theta'}{r^2} \tag{2.7}$$

Which is also known as *The fundamental law of photometry*. Also note that we know:

$$\partial\Omega' = \frac{\partial A \cdot cos\theta}{r^2} \tag{2.8}$$

Yields:

$$\partial\Phi = L_{e,\Omega} \cdot \partial A' \cdot cos\theta \cdot \partial\Omega' \tag{2.9}$$

Therefore, for receiving and emitting patches very similar formula applies, for graphical representation see 2.2.

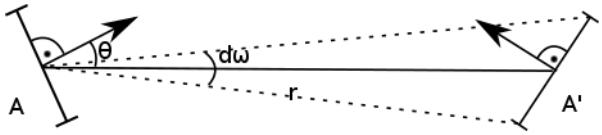

Figure 2.2: Fundamental law of photometry, image representation, A and A' are the surfaces positioned in distance of r.

2.1.3 The behavior of light

To construct an image, the computation of light reaching the camera has to be performed. In case of real physics, the simulation should start by emitting the light from light emitting surfaces. Followed by the light scattering simulation and gathering the light arriving to the camera.

The problem here is, that most of the light will not scatter into the camera, but ends up 'not' hitting the camera at all. This is optimized by flipping the direction.

The simulation starts in the camera point, light scatter into the scene and the energy is contributed each time we hit the light.

It is important to note, that light (upon hitting the surface) can undergo either:

- **Reflection**

  Light is reflected, obeying laws of reflection (see image 2.4):

  – The incident ray, the reflected ray and the normal at the given point lie in the same plane.

  – The angle between incident ray and normal equals the angle between reflected ray and normal.

  – The incident ray and the reflected ray are on opposite sides of normal.

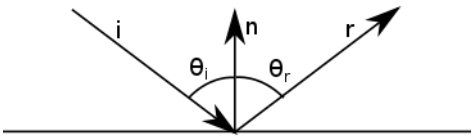

Figure 2.3: Reflection of incident ray i at surface with normal n into ray with direction r. Note that incident angle θ_i equals the reflected angle θ_r.

In equation:

$$\mathbf{R} = \mathbf{I} - 2(\mathbf{I} \cdot \mathbf{N})\mathbf{N} \tag{2.10}$$

Where:

– $\mathbf{R}$ is a vector representing direction of reflected ray.

– $\mathbf{I}$ is a vector representing direction of incident ray.

– $\mathbf{N}$ is surface normal at given point.

– operator $\cdot$ represent scalar inner product between vectors.

Note that for diffuse surfaces we often talk about diffuse reflection, in reality this is a reflection on the surface of the object. Although, the surface is not perfectly smooth but its surface is highly irregular. The light reflection on such surface is diffuse reflection, where reflected ray directions appear to be random.

- **Refraction**

  Light is refracted, obeying Snell's law of refraction (see 2.4). In equation:

$$\frac{sin\theta_1}{sin\theta_2} = \frac{n_1}{n_2} \tag{2.11}$$

  Where:

  – θ_1 is an angle between incident ray and surface normal.

  – θ_2 is an angle between refracted ray and opposite vector to surface normal.

  – n_1 is refractive index in first medium.

  – n_2 is refractive index in second medium.

  Note that for computing refracted direction the vector form of this equation is used:

$$\mathbf{R} = \frac{n_1}{n_2}\left[\mathbf{N} \times (-\mathbf{N} \times \mathbf{I})\right] - \mathbf{N}\sqrt{1 - \left(\frac{n_1}{n_2}\right)(\mathbf{N} \times \mathbf{I}) \cdot (\mathbf{N} \times \mathbf{I})} \tag{2.12}$$

  Where:

  – **N** represents surface normal.

  – **I** represents incident direction.

  – n_1 is refractive index in first medium.

– n_2 is refractive index in second medium.

Further, to describe the behavior of light when crossing between two media with different refractive index, Fresnel equations have to be introduced. In the following approach Fresnel factor in the specular reflection of light is used, for which there exists Shlick's approximation [9].

$$R = R_0 + (1 - R_0)(1 - \mathbf{H} \cdot \mathbf{V})^5 \tag{2.13}$$

Where:

– R_0 is reflection coefficient for incoming light that is parallel to normal at given position. Defined as (n_1 and n_2 represents index of refraction for given media):

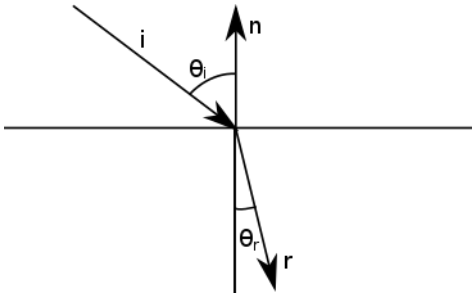

Figure 2.4: Refraction of incident ray i at surface with normal n obeys Snell's law of refraction.

$$R_0 = \left(\frac{n_1 - n_2}{n_1 + n_2}\right)^2 \tag{2.14}$$

– **H** represents half angle vector - halfway between viewing vector and incident light vector.

– **V** represents viewing vector.

- **Absorption**

  Light stops at the surface and the light does not reflects or refracts.

The search for each incoming light particle directed towards the camera is therefore straight forward for perfectly reflective and refractive surfaces (as they have to obey laws of reflection and Snell's law of refraction), although for diffusely reflective surfaces, as mentioned, is non-trivial due to random behavior of the particle. Such random behavior is modeled by means of probability theory, defining the density function as:

$$r(x, \Omega', \Omega) \cdot \partial\Omega = p \tag{2.15}$$

Where:

- x represents given point in space where light particle arrives.
- Ω' represents the direction from which the light particle arrives.
- Ω represents the direction into which the particle is scattered.
- p determines the probability of scattering the particle from direction Ω' into direction Ω.

Note that for perfectly reflective or refractive surfaces such function can also be constructed (resulting in probability equal to 1 under single direction obeying the laws of reflection, respectively Snell's law of refraction, and 0 otherwise).

2.1.4 The derivation

The energy reaching the camera is, for given surface, described by its light emission (the direct component) and light scattered into the camera (the indirect component).

The indirect component is calculated as following:

$$\Phi_o = \Phi_e + \int \int_\Omega (r(x, \omega', \omega) d\omega) \Phi_i d\omega' \tag{2.16}$$

Using specular radiance instead of flux yields:

$$L(x, \omega) = L_e(x, \omega) + \int_\Omega L_i(x, \omega') \frac{r(x, \omega', \omega)}{cos\theta} cos\theta_{\omega'} d\omega' \tag{2.17}$$

Where:

- $L(x, \omega)$ Represents spectral radiance arriving into the camera from given position x.
- $L_e(x, \omega)$ Is emitted light at given point x into the camera.
- $L_i(x, \omega')$ Is incoming light to given point x from direction ω'.
- $\frac{r(x,\omega',\omega)}{cos\theta}$ Represents Bidirectional Scattering Distribution Function - BSDF (the basic one is BRDF - Bidirectional Reflectance Distribution Function)[1].

This equation is also known as the rendering equation, by solving this equation for each visible point an image is produced.

2.2 Monte-Carlo Integration

Equations, like the rendering equation, tend to be very hard for analytic solving, due to high order integrals of large functions. For computing such equation it is crucial to use generic technique for integral computation, like Monte-Carlo integration.

Monte-Carlo integration is one of the Monte-Carlo methods (more precisely random walk Monte-Carlo method), that repeatedly run

1. BRDF describes how the light behaves (reflects/refracts) when reaching the surface.

the simulation with random samples, obtaining numerical results. Monte-Carlo integration is a strong tool for solving definite integrals.

Given one dimensional integral equation:

$$I = \int_a^b f(x)\mathsf{d}x \tag{2.18}$$

Due to the law of large numbers, it is well known that this integral equation can be estimated using:

$$Q_N = \frac{b-a}{N}\sum_{i=1}^{N} f(X_i)p(X_i) \tag{2.19}$$

Where:

- N Represents the total number of samples taken by Monte-Carlo estimator
- X_i Represents i-th randomly generated value in range $[a, b]$. The random number generator needs to have probability distribution function p.

By examining the previously derived rendering equation, it is possible to determine that on the right part the integration is performed also over the same function we're searching for on the left (yet with different arguments). Such equations are known to be Fredholm's equations of the second kind.

Solving such equation leads to expansion into Neumann's series:

$$L = \sum_{i=0}^{n} \Gamma^i L_e + \Gamma^{n+1} L \tag{2.20}$$

Where:

- $\Gamma^i L_e$ represents the light arriving into camera gathered from emissive surfaces using i-th bounce (0-th bounce represents directly emitted light into camera -> e.g. visible light source); going up for n bounces.

- $\Gamma^{n+1}L$ represents the rest of the light arriving into camera, gathered after n bounces.

Due to contractive property of the equation (as it is known that with each bounce the energy decreases):

$$\lim_{n->\infty} \Gamma^{n+1}L = 0 \tag{2.21}$$

And so, it can be rewritten as Liouville-Neumann's series:

$$L = \sum_{i=0}^{\infty} \Gamma^i L_e \tag{2.22}$$

This leads to high dimensional integral equations. These equations can be effectively solved using random walk method, such as Monte-Carlo integration.

For high dimensional integral equations, Monte-Carlo integration is defined as:

$$\int_{a_0}^{a_1}\int_{b_0}^{b_1} f(a,b)dadb = \frac{(a_1-a_0)(b_1-b_0)}{N}\sum_{i=1}^{N} f(X_i); X_i \in ([a_0,a_1],[b_0,b_1]) \tag{2.23}$$

$$\int_{a_0}^{a_1}\int_{b_0}^{b_1}...\int_{z_0}^{z_1} f(a,b,...,z)dadb...\mathsf{d}z =$$
$$= \frac{(a_1-a_0)(b_1-b_0)...(z_1-z_0)}{N}\sum_{i=1}^{N} f(X_i); X_i \in ([a_0,a_1],[b_0,b_1],...,[z_0,z_1]) \tag{2.24}$$

The last problem remaining in this technique is the fact, that the computation of Liouville-Neumann series needs infinite number of steps. As the simulation is computing the light, each bounce of the light the energy

decreases, so after some bounces the remaining energy will be of little consequence. As a result it is necessary to determine means for terminating this light path.

A simple method (that still keeps system energy conserving and thus physically correct), is that after the energy of the path drops below some certain level, there is a chance of $p \in [0, 1]$ proportional to the energy of the path to terminate this path, by generating a random value with uniform probability distribution function in interval $[0, 1]$ and comparing to p it is decided whether the current path is terminated or not. This method is also known as **Russian Roulette**.

2.3 Ray Tracing

The assumption of light traveling at infinite speed along straight lines leads to calculating intersections between such lines and scene in general.

Calculating traversal of the ray, originating in camera with given direction, through the 3-dimensional scene determines the first visible point from camera in such direction.

Upon intersection, the ray can be either reflected (according to laws of reflection, note that also diffuse reflection is possible), refracted (according to Snell's law of refraction) or terminated.

2.3.1 Ray-Primitive Intersection

It is critical to correctly determine visibility between two points (e.g. whether there is a primitive blocking line between two points) or to calculate whether there is any primitive along the ray, that is hit by the ray. Such tests are nothing more than solving an equation of ray against an equation of given primitive.

Such intersection test can be computed using either numerical estimation and numeric method (in case of primitives composed of high-order

functions), or by direct analytic solution (preferred, as the scenes are mostly composed of simple primitives like spheres, triangles, axis-aligned bounding boxes, etc.).

One of the goals, when taking also the final algorithm performance into account, is to decrease the number of kinds of geometric primitives defining the scene. Each primitive can be well described using triangular geometry, and divided into sections using axis aligned bounding boxes. No more primitive types are needed and so the provided theory is only for intersection between ray and these two.

Primitive Definitions

Before the actual description of intersection algorithms, a definition of all the primitives has to be provided.

Ray

A ray is infinite half-line beginning in given point, pointing into specific direction. The equation of ray is:

$$\mathbf{x} = \mathbf{o} + t \cdot \mathbf{d}; t \in [0, \text{inf}] \tag{2.25}$$

Where:

- $\mathbf{o}$ is a 3-dimensional point holding the coordinates of ray origin.
- $\mathbf{d}$ is a 3-dimensional unit vector defining the ray direction.
- t is a parameter belonging to interval $[0, \text{inf}]$ and thus defining the bounds of the given ray.

Triangle

A triangle is basic geometric primitive typically defined by 3 points and 3 barycentric coordinates (See 2.5).

$$\mathbf{x} = \mathbf{a} \cdot \alpha + \mathbf{b} \cdot \beta + \mathbf{c} \cdot \gamma \tag{2.26}$$

Where:

- a, b, c are the points defining the triangle.
- α, β, γ are barycentric coordinates, for which: $\alpha \in [0, 1]$, $\beta \in [0, 1]$, $\gamma \in [0, 1]$ and $\alpha + \beta + \gamma = 1$.

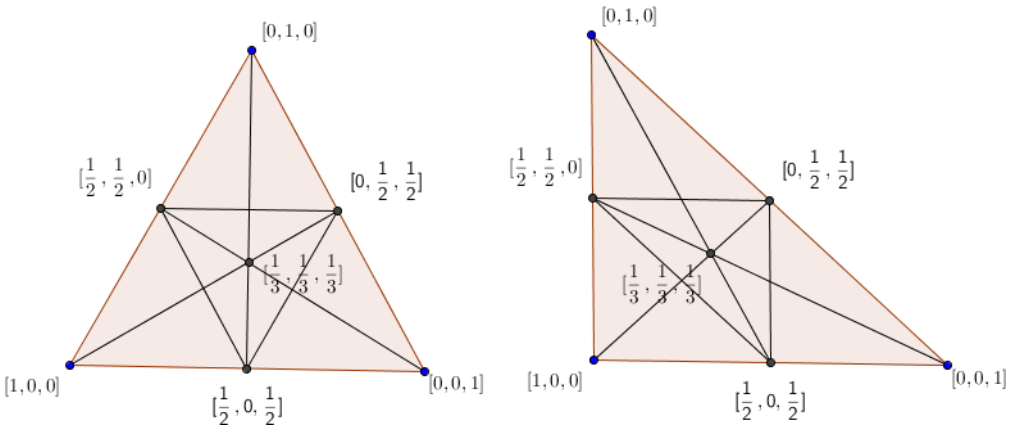

Figure 2.5: Shows barycentric coordinates at various points on an equilateral triangle and a right triangle.

Axis-Aligned Bounding Box

Axis-Aligned Bounding Box is defined as a volume enclosed by 6 planes (2 planes per axis) in a space. As these planes are axis-aligned they can be defined by using just a single value. The definition is thus:

$$\mathbf{n} = (n_x, n_y, n_z), \mathbf{f} = (f_x, f_y, f_z); n_x < f_x, n_y < f_y, n_z < f_z \tag{2.27}$$

Where the volume lies between n_x and f_x on x axis, n_y and f_y on y axis, n_z and f_z on z axis.

2.3.2 Intersection Algorithms

Ray-Triangle Intersection

The intersection is defined as the solution of system described by a ray equation and a triangle equation. By solving for barycentric coordinates it is possible to compute the hit point of the intersection, see image 2.6.

As the performance of such test has major impact on solving the rendering equation, it is critical to decrease the resources used during the computation, the following modification is used (as in Woop et. al. [13]).

For every non-degenerate triangle it is possible to find a transformation that transforms this triangle into a unit triangle [2] (note that such transformation is preserving points, lines, planes and ratios of distances, e.g. it is affine transformation.).

For triangle T, defined by three points $\mathbf{a}, \mathbf{b}, \mathbf{c}$, with normal defined as:

$$N = \frac{(\mathbf{a} - \mathbf{c}) \times (\mathbf{b} - \mathbf{c})}{|(\mathbf{a} - \mathbf{c}) \times (\mathbf{b} - \mathbf{c})|} \tag{2.28}$$

2. A unit triangle is a triangle of size equal to one, that is lying on X and Y plane

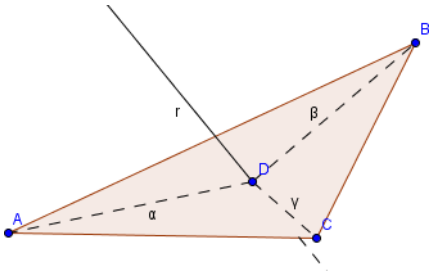

Figure 2.6: Shows a solution of ray-triangle intersection, intersection happens in point D with barycentric coordinates α, β, γ.

The inverse transformation for the wanted one is defined as:

$$M^{-1}(T_{unit}) = \begin{pmatrix} a_x - c_x & b_x - c_x & N_x & c_x \\ a_y - c_y & b_y - c_y & N_y & c_y \\ a_z - c_z & b_z - c_z & N_z & c_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.29}$$

The wanted matrix M can be thus computed by inversion of this matrix.

The intersection is performed by transforming the ray into the space of such unit triangle and performing the test there. As the transformation is affine, the barycentric coordinates are equivalent (for the case before and after transformation). The computation is:

$$t = \frac{M_{1,4} - o_x M_{1,1} - o_y M_{1,2} - o_z M_{1,3}}{d_x M_{1,1} + d_y M_{1,2} + d_z M_{1,3}} \tag{2.30}$$

$$\alpha = M_{2,4} + o_x M_{2,1} + o_y M_{2,2} + o_z M_{2,3} + t \cdot (d_x M_{2,1} + d_y M_{2,2} + d_z M_{2,3}) \tag{2.31}$$

$$\beta = M_{3,4} + o_x M_{3,1} + o_y M_{3,2} + o_z M_{3,3} + t \cdot (d_x M_{3,1} + d_y M_{3,2} + d_z M_{3,3}) \tag{2.32}$$

Where:

- $\mathbf{o} = (o_x, o_y, o_z)$ represents non-transformed origin of ray.
- $\mathbf{d} = (d_x, d_y, d_z)$ represents non-transformed direction of ray.
- t is a distance from ray origin towards the hit point along ray direction.
- α, β are barycentric coordinates.

The intersection occurs, when all of the following statements are true for given intersection test:

1. $t > 0$ distance t towards the intersection is non-negative.
2. $\alpha \geq 0 \wedge \alpha \leq 1$ and $\beta \geq 0 \wedge \beta \leq 1$ both computed barycentric coordinates are in respective bounds.
3. $\alpha + \beta \leq 1$ both barycentric coordinates summed together are less than 1 (meaning $\gamma >= 0 \wedge \gamma <= 1$).

The first statement defines whether there is an intersection between ray and triangle plane at distance t. Followed by two statements determining whether the hit point lies inside triangle bounds.

This test is highly efficient when the matrix can be pre-computed, as the number of operations that need to be performed for single intersection test is minimal compared to other intersection tests.

Ray-Axis Aligned Bounding Box Intersection

As in previous case, the intersection occurs whenever there is a solution for system of equation composed of both, ray and axis-aligned bounding box equation. It uses so called slab test (Simplified 2D equivalent in image 2.7).

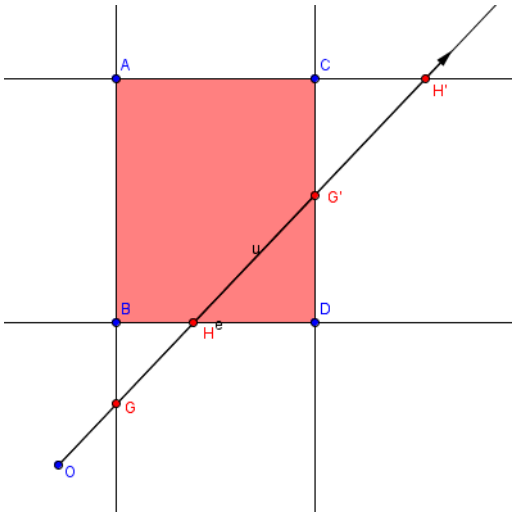

Figure 2.7: 2D Intersection of ray (defined by O and direction vector) and Axis-Aligned Bounding Box (defined by A, B, C, D) calculates hit points (G, H, G', H') using slab test, which determines whether there is any intersection.

A slab is an interval defined between 2 values, and undefined elsewhere. Transforming the definition of axis-aligned bounding box from 6 planes to 3 slabs, 3 value ranges where the box is defined are obtained, each for separate axis. Each of such slabs is defined by 2 planes - minimum and maximum bounding plane. For the purpose of high efficiency intersection, the slab test is performed, the following text describes its derivation according to Ericson et. al. [5].

For each slab two intersections points with the ray can be computed - the entry point and the exit point, as:

$$t_1 = \frac{n_k - o_k}{d_k}, t_2 = \frac{f_k - o_k}{d_k} \tag{2.33}$$

Where:

- $i = min(t_1, t_2)$ represents the entry point.
- $o = max(t_1, t_2)$ represents the exit point.

Finding both, entry and exit points, for all slabs is then:

$$t_{enter} = max(i_0, i_1, ...i_k), t_{exit} = min(o_0, o_1, ...o_k) \tag{2.34}$$

Where k represents k-th dimension of computation, for the bounding box case $k = 3$.

The intersection occurs when $t_{exit} > 0$ and $t_{enter} < t_{exit}$.

2.3.3 Acceleration Structure

Even though the previously defined intersections are efficient, it would mean a lot of intersection tests when using naively computed ray tracing. For a simple scene with 100 000 triangles and standard FullHD resolution, this would effectively mean performing 207 360 000 000 intersection tests in total for single bounce, which is way too large number, while most triangles are not even near the part of the scene through which the ray travels.

The idea of acceleration structure is following, by creating a structure for the scene, keep small groups of primitives in nodes (where all primitives in a node are close to each other in terms of position). When searching for intersections between the ray and triangles, we first determine the nodes in acceleration structure that intersect our ray and then intersect the triangles in these nodes.

Also note that as the search is only for the closest hit, this can be even further sped up by searching the hierarchy and thus the scene along the ray direction, beginning at the origin, going forwards in direction.

The performance of acceleration structure can be further increased by creating a hierarchical structure (most common is tree hierarchy) of nodes, where interior nodes contain other nodes (smaller) and leaves contain actual geometry.

Bounding Volume Hierarchy

The bounding volume hierarchy (BVH) is N-ary tree structure. The most common is a binary tree structure, although for some hardware, using the tetrary tree structure (also denoted as QBVH) was also found to be efficient, like Dammertz et. al. [4]. For the following text, I refer to BVH as to a binary tree structure (For example of BVH see figure 2.8).

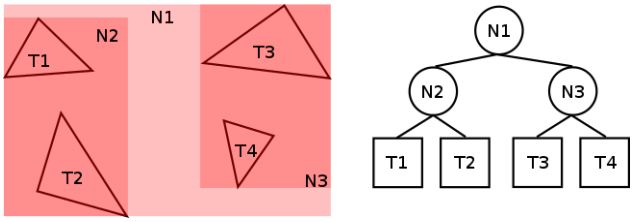

Figure 2.8: Bounding volume hierarchy built on 4 triangle primitives (T_1, T_2, T_3, T_4), spatial representation on the left and tree representation of the data on the right. N_1, N_2, N_3 are the bounding volumes of three nodes.

For BVH there are two important parts, the part where the acceleration structure is built and where it is traversed by a ray. As the implementation in this thesis focuses mainly on run time performance, the time spent building the hierarchy is not further considered.

Build

Building of BVH is often recursive algorithm similar to any other tree building algorithms (recursively builds the leaf nodes first, later creating interior nodes):

1. Compute the extents of the scene and create bounding box of whole scene.

2. Recursively: If the primitive set currently processed is small enough, create a leaf node, compute its bounding box and return. Otherwise, find a position where the primitive set is divided

into two subsets (using some heuristics) and create interior node, with two child nodes, both created by recursively applying this algorithm on both subsets, the bounding box of such node is a bounding box around of both child nodes bounding boxes.

The division into two subsets determines what will be the quality of the resulting BVH. There are multiple approaches, like:

- **Surface area heuristics** is an approach, where in general for each interior node there is a cost computed (based on the surface areas of the objects) that is heuristically minimized. Effective algorithm is described in Pharr et. al. [8].
- **Bounding Volume Hierarchy with Spatial Splits** is a hierarchy, that was build in a similar way to surface area heuristics based one. Although instead of considering just the start point and the end point of each primitive, some possible splits of each primitive are also considered, as in Stich et. al. [10]. It produces one of the best quality trees.

Traversal

Traversal through any BVH can be done in a generic way, using standard tree searching algorithm, an example in pseudo-code is provided in algorithm 1.

These stack-based traversals first search for leaf nodes, those contain triangles. Once we find a leaf node, the intersection against all triangles in leaf node is performed. Later, pop next node from stack and continue with traversal.

Algorithm 1 BVH traversal

```
1:  procedure TRAVERSE
2:      node ← root
3:      stack ← empty stack
4:      while node ≠ null do
5:          while node is interior do
6:              test ray with both child AABBs
7:              if ray hits single child node then
8:                  node ← intersecting child
9:              else
10:                 if ray hits both child nodes then
11:                     node ← closer child
12:                     stack.push(further child)
13:                 else
14:                     node ← null
15:             if node = null then
16:                 node = stack.pop()
17:         for all items in leaf node do
18:             intersect ray vs triangles, store closest result
19:         node = stack.pop()
```

2.3.4 Image Plane

The results of the computation are stored in an image plane. The image plane is a matrix of $W \times H$ size, each item in this matrix is a three component vector $C = (R, G, B)$.

These components shows red, green and blue color of given item, each of this item therefore represents a pixel. The value of each single component represents the intensity in range $[0, 1]$.

For each of the pixels it is possible to compute a ray direction from the camera point into the scene, passing through the pixel, this direction

represents the first direction cast from camera viewpoint into the scene.

2.4 Path Tracing

Having all the pre-requisites defined, the following section describes proof-of-concept algorithm on solving the rendering equation. The following text gives brief overview of the approach.

First of all there is an assumption of having the scene and data structure (BVH) prepared, this means that there is a virtual camera defined in the scene, from this camera position an image will be generated. The computation itself generates N samples per each pixel, this means that the rendering equation is sampled N times and the results are appropriately weighted.

Algorithm 2 Path Tracing

```
1:  procedure PATHTRACE
2:  for each path:
3:      ray ← setup primary ray
4:      while ray.terminated = false do
5:          result ← raycast(ray)
6:          if result.hit = false then
7:              Accumulate background color
8:              ray.terminated ← true
9:          else
10:             Compute and Accumulate surface emission
11:             if Russian roulette terminates path then
12:                 ray.terminated ← true
13:             else
14:                 ray ← Get B*DF Sample
```

- For each pixel in image plane I do:
- For 1...N:
  1. Generate a ray from camera in direction of currently processed pixel.
  2. Recursively (note that each recursive step computes single bounce of the light):
     Compute closest intersection on given ray with the scene. In case of no intersection, return black (zero intensity). Compute the light emission x at the hit point.
     If the contribution of current bounce (i-th bounce) is smaller than pre-defined constant k (this constant determines when we start applying Russian roulette), perform Russian roulette and possibly terminate the light path.
     Determine the direction of next bounce (reflect, diffusely reflect or refract), and recursively compute it into value x'.
     Return the light emission at current hit point (e.g. x) summed with x'. [3]
- The resulting color is accumulated for single pixel and at this point divided by the number of samples (e.g. N), producing N-sample approximation of the rendering equation.

For completeness, see also pseudo-code for the algorithm 2 and the reference image 2.9.

As for the correctness of this algorithm, the recursive part of the computation calculates single sample of the Monte-Carlo integration of the rendering equation. These samples are summed and divided by number of samples, resulting in an approximation of the rendering equation.

3. This is nothing more than the actual single sample from Monte Carlo integration of the rendering equation.

Another question is whether this algorithm finishes. All steps, with exception for the recursive part clearly finish. As for recursive part, each next bounce, the light contribution decreases, hence the Russian roulette terminates the path sooner or later.

The convergence rate with pseudo-random number generator (simulating real random number generator) is: $\frac{1}{\sqrt{N}}$.

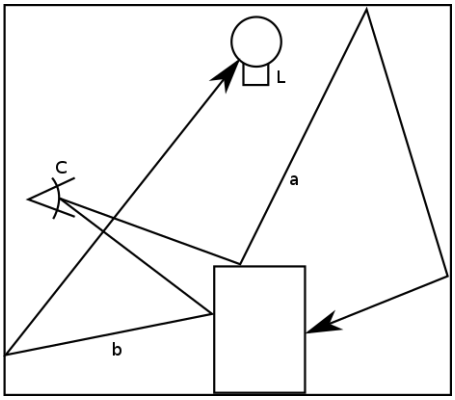

Figure 2.9: From camera C two paths, a and b are traced into the scene with one light L. The path a has zero contribution as it does not hit any emissive surface, while the path b has non-zero contribution, hitting the L after two bounces.

2.5 Improvements of Path Tracing

While the previous code indeed computes the image according to the rendering equation it is highly ineffective due to light paths getting terminated before they actually hit any surface with emission energy. Therefore the energy contribution of such light paths is zero.

Also, when uniformly choosing directions, a direction direction with almost 0 effect on the resulting energy is selected (e.g. selecting the directions that are almost zeroed out by BSDF).

Thus, rendering direct and indirect caustics along with hard case

scenarios takes too many samples with standard path tracing approach.

2.5.1 Explicit Sampling

A typical problem with standard path tracing are small light sources. The probability of hitting infinitely small light sources (e.g. point lights) is equal to 0.

When examining the rendering equation, it is clearly visible that the integration goes over the whole hemisphere above the hit point. If there is any directly visible light in the hemisphere, there is a way to add its contribution to the path without breaking the unbiased nature of the algorithm.

Given the rendering equation (recall its derivation in Section 2.1.4):

$$L(x,\omega) = L_e(x,\omega) + \int_\Omega L_i(x,\omega')\frac{r(x,\omega',\omega)}{cos\theta} cos\theta_{\omega'} d\omega' \tag{2.35}$$

It is possible to calculate direct emissive light from the hemisphere above the hit point as:

$$L'_e(x,\omega) = \int_\Omega V(x,\omega')\frac{r(x,\omega',\omega)}{cos\theta} cos\theta_{\omega'} d\omega' \tag{2.36}$$

Where:

- $V(x,\omega')$ represents visibility function between x and first hit in direction of ω'. Such function returns 0 when non-light surface is hit, otherwise yields the color of the light.

This contribution can be directly added into the rendering equation, yet as the emission is sampled explicitly inside the integral part of the equation, the emission energy must not be added when hitting the surface with some light emission (so that we do not add the same contribution twice), hence removing original L_e from the equation is a must.

As for the implementation, this modification is straight forward. During the recursive light path computation the light emission contribution is not added, instead:

1. Select random surface with non-zero light emission, and generate a random position on this surface (this is the light position that is explicitly sampled)

2. Calculate visibility function and weight it according to the equation 2.36 - calculating L'_e

3. Instead of returning the emission on current iteration, return explicitly computed light emission summed with results of the next iteration.

While this modification is straight forward, it has large impact on scenes with direct light (or easy-to access light sources). For reference, see image 2.10.

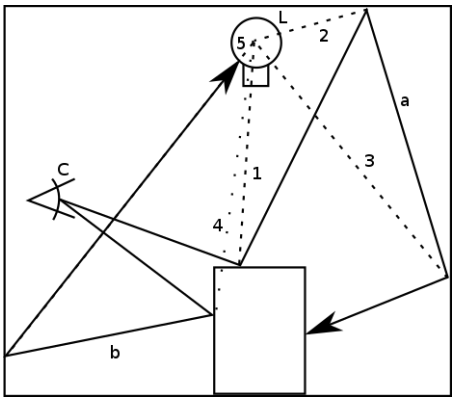

Figure 2.10: Both paths, a and b explicitly sample light each step. Dashed explicit samples directly hit light and therefore add energy to light contribution of the path $(1, 2, 3, 5)$, one dotted path does not have direct visibility to the light and therefore at that step, there is zero contribution (4).

2.5.2 Importance Sampling

Another improvement with large impact on the result is related to a way how reflected rays and refracted rays are generated.

Selecting random ray directions (with uniform probability distribution function) most likely ends up also selecting directions that are almost orthogonal to a surface normal, the light contribution of such paths tends to have small impact on resulting energy.

Instead of generating rays with uniform probability, they should be generated in a weighted manner (Monte-Carlo integration allows for that, as described previously). The correct weighting for perfectly diffuse surfaces is cosine-weighting (See 2.11). With cosine-weighted ray generation the rendering equation is modified to:

$$L(x,\omega) = L_e(x,\omega) + \int_\Omega L_i(x,\omega')\frac{r(x,\omega',\omega)}{cos\theta}d\omega' \tag{2.37}$$

Note that the importance sampling is actually already done for perfect reflection and perfect refraction, as they are always selecting the only direction that can generate energy contribution (any other direction would end up with contribution equal to zero).

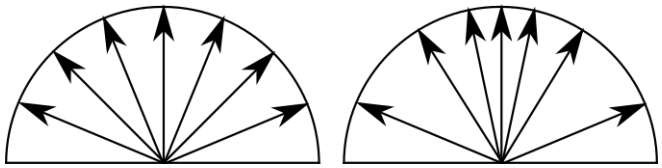

Figure 2.11: Vectors generated with uniform weighting on the left, with cosine-weighting on the right.

2.5.3 Bi-Directional Path Tracing

While the previously presented modifications improve convergence rate and thus the performance of the algorithm, they still do not solve the actual problem with caustics and hard cases.

The idea behind bi-directional path tracing builds on explicit sampling, further extending it according to Veach et. al. [11]. Bi-directional path tracing is a two-pass algorithm:

1. Light path generation

   First phase starts by generating a random point on any of the light sources in the scene. From this point a path into the scene is created, storing all the hit points with energy information. So after N bounces, there is $N + 1$ vertices (one more vertex for first, generated point on light).

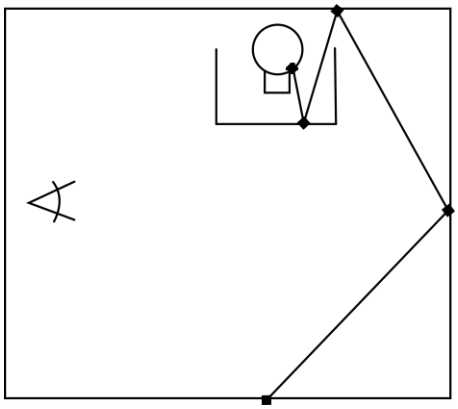

Figure 2.12: The generated light path, stored vertices on the light path are marked with black dot.

2. Camera path generation and path join

   Second phase follows similar pattern as standard path tracing. Starting off from camera position, a path is traced into the scene. Each hit point there is a possibility to calculate visibility to zero-or-more vertices on the light path, adding energy contribution to the camera path (See 2.13).

   The connection of both paths (e.g. the part where we compute visibility between single vertex on camera path to zero-or-more

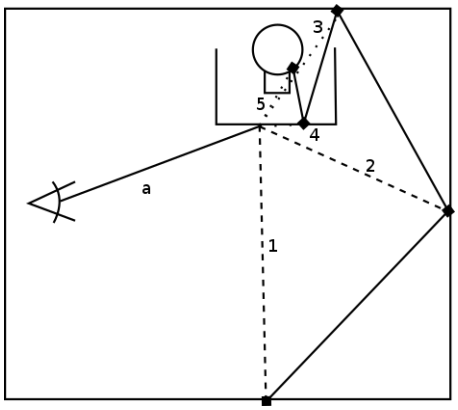

Figure 2.13: The camera path a in a full join with light path. Connections $1, 2$ have non-zero contribution for the camera path, connections $3, 4, 5$ have zero contribution. Note that the scene represents typical hard case for path tracing.

vertices on the light path) is also called path join. There are multiple ways to perform path join:

- Full join

  The most intuitive solution is: For each point on the camera path the join is performed against all the points on the light path.

  While this approach improves the quality in a single sample computation the most, it also needs the most resources.

- Single-step join

  Performance wise the most interesting approach, the join is performed only one-to-one on the endpoint of the light path to the endpoint of the camera path.

  While this provides even worse convergence rate than explicit sampling, it does not need any more memory or computation power compared to standard path tracing (without explicit sampling).

- k-step join

  On each vertex on the camera path the algorithm selects k random vertices on the light path. The visibility test is performed against these vertices.

  This approach provides convergence rates comparable to full join, while the number of actual visibility computations is smaller.

The first approach can be computed interactively per each computation of the second pass (e.g. single sample computes single light path and single camera path), it can also be computed once per multiple computations of the second pass. Also it is possible to pre-compute the first pass and re-compute on demand.

For the comparison, the image 2.14 shows the difference between the path tracing with explicit sampling and bi-directional path tracing, both images took the same computation time. Also, for the comparison, the pseudo code for bi-directional path tracing is provided in algorithm 3.

Algorithm 3 Bidirectional Path Tracing

```
1: procedure BIDIRPATHTRACE
2: for each path:
3:     // Generate light path
4:     Generate vertex on random light
5:     Push this vertex to light path
6:     ray ← setup light ray
7:     while ray.terminated = false do
8:         result ← raycast(ray)
9:         if result.hit = false then
10:            ray.terminated ← true
11:        else
12:            Push this hitpoint to light path
13:            if Russian roulette terminates path then
14:                ray.terminated ← true
15:    // Trace camera path
16:    ray ← setup primary ray
17:    while ray.terminated = false do
18:        result ← raycast(ray)
19:        if result.hit = false then
20:            Accumulate background color
21:            ray.terminated ← true
22:        else
23:            Compute contribution by joining light path
24:            with this vertex of camera path
25:            if Russian roulette terminates path then
26:                ray.terminated ← true
27:            else
28:                ray ← Get B*DF Sample
```

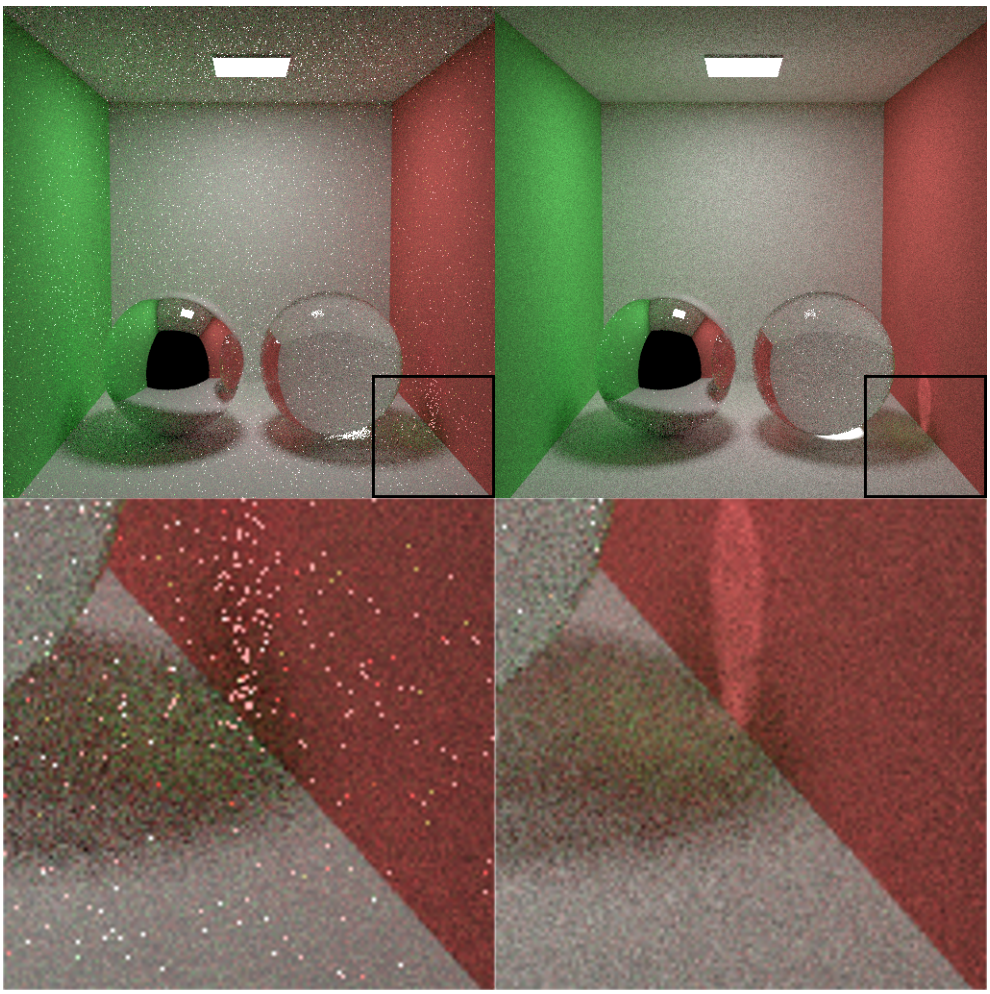

Figure 2.14: Top row of the images shows the comparison between path tracing with explicit sampling and bi-directional path tracing for approximately same time. Bottom row of the images shows the same comparison with 4-times magnification of the area inside the black rectangle. Notice the quality difference in caustics.

3 Massively Parallel Hardware Architecture

The following chapter provides information about target massively parallel hardware architecture (e.g. the graphics processing unit) with detailed description of it from low-level and high-level point of view.

The graphics processing unit is a high-latency high-throughput compute device used for executing large number of threads in parallel. The graphics processing unit (also named the device in this context) should work as another processor working next to main central processing unit (also named the host in this context).

In general, data-parallel and compute intensive parts of the application can be off-loaded from the host to the device. Such functions (that are data-parallel and executed many times) can be isolated into a function (kernel function, such function is applied to each element in the stream), which can be executed on the graphics processor unit.

3.1 Hardware Architecture

3.1.1 Stream Multiprocessor

Modern graphics processing unit micro-architecture is built around an array of Streaming Multiprocessors (also denoted as SM). The streaming multiprocessor performs the actual computation, it has its own control unit, registers, execution pipelines and caches, see 3.1.

Streaming multiprocessor is composed of (details are given according to Fermi architecture, described in Witterbrink et. al. [12], GPUs based on this architecture were used to measure results in this thesis):

- CUDA Cores

  These cores perform the actual computation for single thread. These are composed of the Floating Point Unit (IEEE 754-2008 floating point standard), the Integer Unit, the Logic Unit and the Branch Unit. Note that they also contain Fused Multiply-And-add (FMA) for single and double-precision operations.

  In case of Fermi, there are 32 CUDA cores per single SM, allowing for 32 32-bit floating point (single precision) operations per clock, 16 64-bit floating point (double precision) operations per clock or 32 32-bit integer operations per clock.

- Warp Schedulers

  These allows for instruction level parallelism. At every instruction issue time, each warp scheduler selects a warp of threads and issues multiple instructions for this warp on cores.

  Fermi has two warp schedulers and issues two instructions per each warp at every instruction issue time.

- Special Function Units

  Special Function Units are used for computing some of the functions like *cos*, *sin*, *log*, *exp*, etc. These units operate as single-precision only (double-precision functions are emulated using multiple instructions), some accuracy loss might be introduced by their usage.

- L1 Cache and Shared Memory

  Per each SM there is 64 KiB of memory dedicated for shared memory and L1 cache.

  L1 cache memory is hardware managed, for Fermi architecture either 16 KiB or 48 KiB can be chosen (by the application).

Aggregate bandwidth per graphics processor unit is 1.03 TiB/s (Fermi).

Shared memory is user managed. Fermi allows to select either 16 KiB or 48 KiB, aggregate bandwith per graphics processor unit is 1.03 TiB/s (Fermi).

- Registers

  Fermi architecture has 32768 32-bit registers per single streaming multiprocessor.

3.1.2 Global Memory

The global memory for graphics processor unit is an equivalent of what is the random access memory (RAM) for central processor unit.

This memory is accessible by both, the central processor unit and the graphics processor unit.

Central processor unit can access this memory by copying from its memory into global memory of given graphics processor unit, or reading back. The data are transferred through PCI-E (PCI-Express) socket - the peak bandwidth is thus approximately 16 GiB/s in each (read and write) direction, this counts for PCI-E version 3.

The size of global memory is in range of 2 GiB to 16 GiB, where bandwidth for the access from graphics processor unit is up to 150 GiB/s (NVidia Quadro series - Fermi based).

Accessing the memory from the graphics processor unit is cached through L2 cache (512 KiB for Fermi). Further operations are cached either through L1 cache (for read and write operations), or texture caches (for read-only operations).

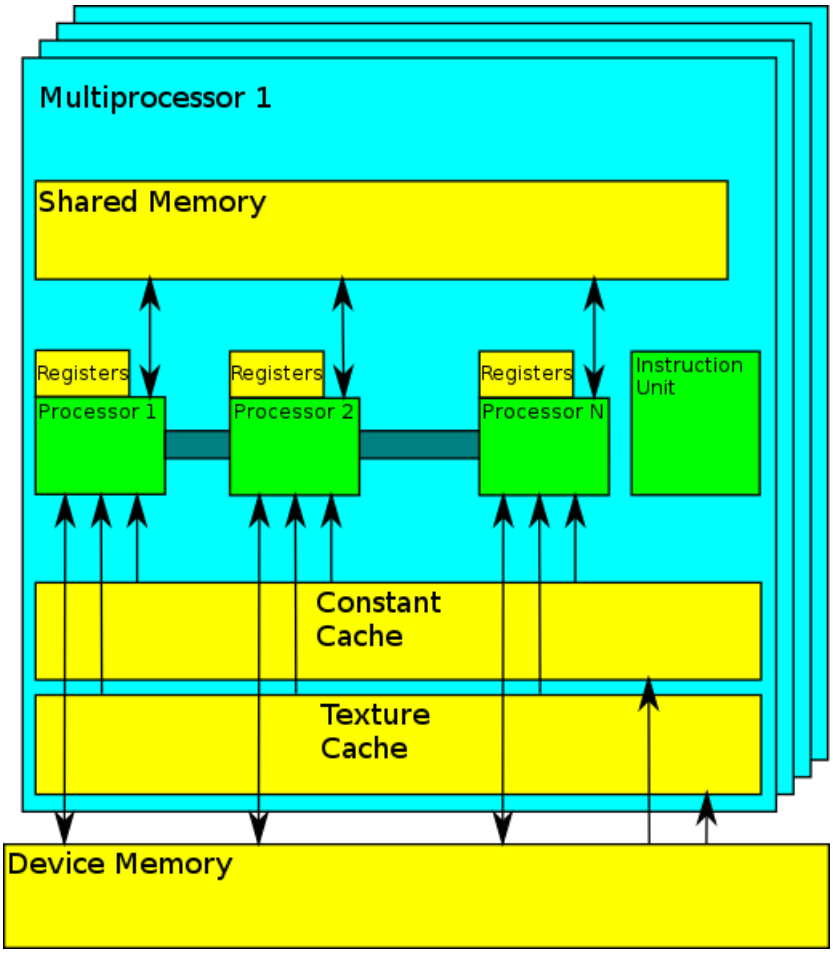

Figure 3.1: Simplified graphical representation of the device and stream multiprocessor.

3.2 Low-level Model

The following section describes further detail about low-level execution of parallel code.

3.2.1 Thread Hierarchy

Threads that are currently executing a kernel are organized in a batch. These batches are further organized as a grid of cooperative thread arrays. These cooperative thread arrays implement CUDA thread blocks.

Threads inside cooperative thread arrays execute in Single-Instruction, Multiple-Thread fashion (SIMT) in groups. These groups are designated as warps.

Single threads inside warp are numbered in sequential order, the number of threads inside warp, also designated as warp size, is hardware dependent constant (typically 32 threads in single warp).

The communication inside cooperative thread arrays can be performed using synchronization points, where threads wait until all threads in a specific cooperative thread array arrive at this point.

The thread identifier is a 3 element vector *tid* (elements are tid_x, tid_y and tid_z), these specify the position of the thread inside 1D, 2D or 3D cooperative thread array. Typically thread identifier is used to determine role of the thread, the data are assigned to the thread based on this value and also the output is written to respective memory location based on the identifier.

Single cooperative thread array can contain just limited number of threads. This limitation is overcome as cooperative thread arrays executing the same kernel can be batched into the grid of cooperative thread arrays, each cooperative thread array in such grid has its ID in terms of the grid, denoted as *ctaid*. Threads in one cooperative thread array are not able to communicate with threads in another cooperative thread array inside the same grid.

Kernel is executed as a batch of threads organized as a grid of cooperative thread arrays. Single cooperative thread array can be executed sequentially or in parallel, depending on the platform.

3.2.2 Memory Hierarchy

There are multiple distinguished memory levels forming a hierarchy (See 3.2), these are:

- Per-thread

  Each thread has assigned private memory for its own operations. This memory is hidden for other threads that are being executed.

  The private memory is assigned once thread execution starts and is released once it finishes.

- Per-block

  Shared memory is assigned per each cooperative thread array, all threads in this block can read and write into this memory.

  The lifetime of the shared memory also starts with block assignment and ends when last warp of the block finishes.

- Global memory

  All threads from all cooperative thread arrays can access global memory.

  In the global memory there are multiple memory spaces available: global, constant, texture and surface. Each of them is optimized for different memory usage.

  The global, constant and texture memory spaces are persistent across different kernel launches in a context of single application.

3.3 Compute Unified Device Architecture

Compute Unified Device Architecture (also CUDA) is a parallel computing platform with its own programming model created by NVidia Corporation.

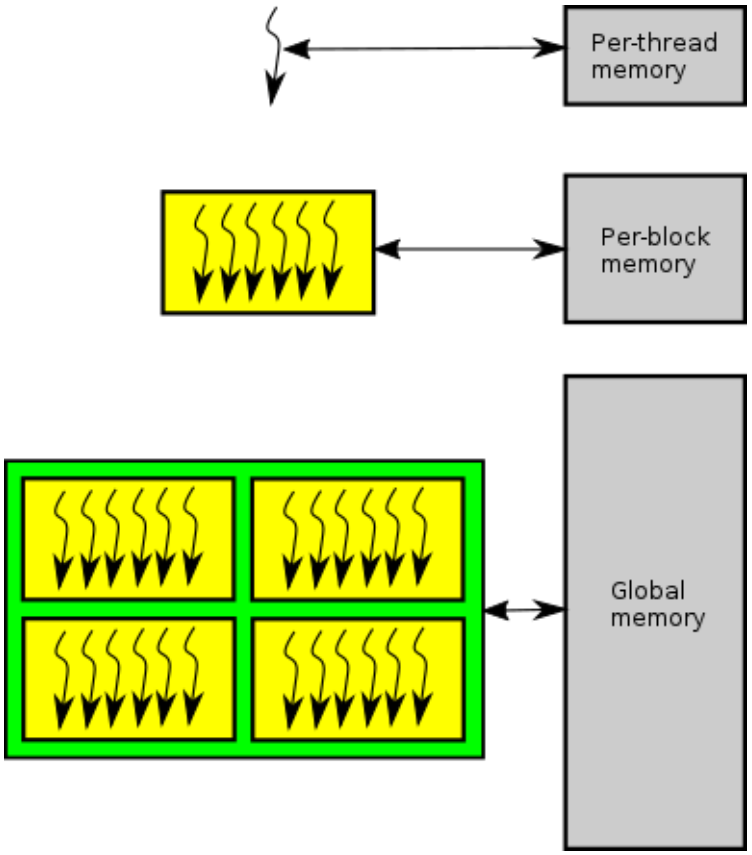

Figure 3.2: Memory hierarchy, showing from where each distinct memory spaces can be reached.

CUDA is implemented by graphics processor units designed and produced also by NVidia Corporation. The computations are performed (on low-level and hardware level) as per description in previous two sections.

It gives the programmers direct access to the instruction set and memory on the CUDA graphics processor units (note that this instruction set is just virtual). Using these, graphics processors can be used for general purpose computations (also denoted as GPGPU - General purpose computing on graphics processor units).

CUDA platform is accessible through extensions to standard programming languages, including C/C++ (the actual compilation of CUDA source code is done through *nvcc* compiler, the language is often designated as C for CUDA). There are also other interface implementations in CUDA platform for: OpenCL (Khronos), DirectCompute (Microsoft), OpenGL Compute Shaders (Khronos) and C++ AMP.

It is possible to inter-operate between CUDA and OpenGL, respectively CUDA and Direct3D, allowing for fast output to the display. The bi-directional path tracer implementation takes advantage of this and allows for real time progressive rendering of scenes.

4 Parallel Bi-directional Path Tracing

The provided implementation is built on top of Timo Aila framework [1], providing a good start point for implementation of complex ray tracing renderers. The following sections gives summary about how the structures to hold the scene data are composed in the implementation and an overview about the actual algorithm implementation on the GPU.

4.1 Data Representation

First of all, a detailed description about how the data are stored in the memory is provided, these information are critical for actual rendering and they have to be available for reading on the GPU during the actual rendering.

A virtual scene is composed of triangles, each triangle has a material assigned to it, all the materials are stored inside a separate buffer. Materials are also connected with textures, which are stored inside single texture atlas. Note that light emissive surfaces are those triangles, whose material has non-zero emissive component. Apart from these data, a virtual camera has to be defined.

Geometry and Materials

The geometry is stored in an indexed manner, e.g. each triangle has 3 vertex indices, thus mapping vertices to the triangles. So in general there are data stored per-vertex and per-triangle.

Per-vertex data:

- Vertex positions

  Storing X, Y and Z coordinate of each vertex. This information is used for generating random positions on triangles (e.g. for

creating starting points of light paths).

- Vertex normal vectors

  Also storing X, Y and Z coordinate of the vector, each normal vector is stored as a unit vector. Used in shading, generating light path directions and also importance sampling.

- Vertex texture coordinates

  Storing U and V coordinates. They are bound in interval $(U, V) \in ([0, 1], [0, 1])$.

Per-triangle data:

- Vertex indexes

  Three vertex indices, the vertices at these locations in vertex buffer represent single triangle.

- Texture atlas data

  Holds offset where texture inside the texture atlas begins and its size. Therefore connects single texture stored in atlas with triangle.

- Material ID

  Determines which material is assigned to given triangle.

- Color

  Storing the color for each triangle is important especially for emissive triangles, as the color defines the actual light color. Also for triangles with no texture, the color is used instead.

For each material there is just a 4-component value stored, describing the reflectivity, the refractivity, the index-of-refraction and the emissivity.

There are two reasons why materials are stored separately:

- Data redundancy

  By definition of material properties per primitive, there would be redundant data per each primitive. Furthermore, most of the definitions would be just a copy of single material properties.

- Storage

  In computer graphics field it is common that artists assign material per whole object. Tools and software for creating models, texturing and animating of the computer graphics scenes follow this pattern and store the data in this fashion.

  The input format for the implementation followed this fashion and the implemented loading software loads all the necessary data from two files - geometry file in 'OBJ' file format and material file in 'MTL' file format:

```
newmtl FloorMaterial
        Ns 0.0000
        Ni 1.5000
        d 1.0000
        Tr 0.0000
        Tf 1.0000 1.0000 1.0000
        illum 2
        Ka 0.0000 0.0000 0.0000
        Kd 0.6000 0.6000 0.6000
        Ks 0.9000 0.9000 0.9000
        Ke 0.0000 0.0000 0.0000
        map_Ka floor_d.tga
        map_Kd floor_d.tga
        emissivity 0.0
        reflectivity 0.0
        refractivity 0.0
        IOR 1.0
```

Where the added properties are *emissivity*, *reflectivity*, *refractivity* and the *IOR* (index-of-refraction). Note that map parameters represent which texture image is used (defined by the file path).

There is one more buffer related to the scene which is used, it is a buffer holding indices to all emissive triangles. This buffer is later used for randomly choosing the starting primitive of the light path.

Note that for intersection routines there is also a buffer for triangles (non-indexed) stored in form of Woop triangles, so two triangle representations are stored. Along with Woop triangles, the intersection routines need acceleration structure. The bounding volume hierarchy stores due to performance reasons both child nodes bounding boxes in parent node. These are stored as three 4-component buffers, e.g.:

- The min_x, max_x, min_y, max_y of the first child node bounding box.
- Stores min_x, max_x, min_y, max_y of the second child node bounding box.
- Stores min_z, max_z, min_z, max_z of both child nodes bounding boxes.

Note that the node data, describing whether the node is a leaf or interior node, along with child node indices (for interior node) or triangle offset and triangle count (for leaf node), are stored in another single 4-component buffer.

4.2 Algorithm Implementation

After the data storage is defined, the algorithmic side of implementation can be described. First of all, as bi-directional path tracing is a multi-pass algorithm which is intended (in this work) to be implemented

fully on the GPU, it is important to describe a way how to execute large functions on the GPU.

Following the definition the actual bi-directional path tracing kernel and its sub-routines are described. As a last section, implementing progressive rendering, allowing for user interaction and real time rendering using bi-directional path tracing.

4.2.1 Mega Kernels

Due to the nature of bi-directional path tracing, there was a need to perform a lot of computations inside single GPU kernel.

In ideal world, the GPU kernel should:

- Generate light path
- Generate camera path, computing the final color and joining against light path
- Do all the post-processing and directly store the results in texture drawn onto screen

Such kernels needed a modification of the framework to allow for doing all the work in a large single kernel, such kernels are also called mega kernels.

Even though the mega kernel approach is not the most efficient way (especially for unidirectional path tracers, as mentioned in Laine et. al. [7]), the amount of data generated in the first phase of bi-directional path tracer required in the second phase is too large to be transferred into another kernel and computed there (in sequential nature the work is performed per single pixel, yet here the work has to be done in parallel for whole screen resolution, hence large amounts of data).

The data of the scene are of course stored inside global memory, local data (the light path) should ideally be held privately per-thread inside cache for fast access. Note that for faster computation, some of

the data are transferred through the texture cache (the texture atlas, triangle representation for intersection).

4.2.2 Traversal

The implementation of acceleration structure allows to use various algorithms of Bounding Volume Hierarchy computation, the traversal algorithm is independent of which of the BVH creation algorithms we use as long as the same layout of the BVH is used, although as the focus was mainly on the performance during the run time, the selected BVH for testing was the Bounding Volume Hierarchy with Spatial Splits.

The actual traversal algorithm is performed according to Aila et. al. [2], in speculative while-while manner. Where each GPU thread performs full traversal for a single ray. Threads in a warp post-pone leaves until all threads found a leaf, then they test intersections against the data in leaves, this guarantees lower execution divergence and thus higher performance.

4.2.3 Bi-directional Kernel

Bi-directional kernel is a mega kernel with input of primary rays and scene description (note that primary rays are generated separately, with stochastic sampling, resulting in anti-aliasing effect), calculating the resulting color after computing N samples per pixel. To keep the description more simple, let assume that only single sample is computed in one execution of the kernel.

The overall structure of mega kernel is as following:

```
Initialize_light_path();
Calculate_light_path();
Calculate_camera_path_with_join();
Store_results();
```

So there are actually four functions, for the last function the detailed description is provided in the following subsection 4.2.4. For the rest of the functions:

- Initialization of the Light Path

  The kernel has input parameters for light path length, the light path generation using Russian roulette would end up in a requirement of dynamic memory allocation which would end up in performance decrease. The path still uses Russian roulette for path termination (although if a low number for maximum path length is selected this property can be broken).

  First, the initialization phase selects a random emissive triangle and a random point on this triangle. Both generated with uniform probability distribution function. This is followed by a generation of initial ray direction, which is done according to importance sampling.

  The resulting data are stored as first vertex of the light path (the position and sampled emissivity, with color of the primitive, of the selected triangle).

  At the end of this section the first vertex of the light path is known, along with ray direction for the next iteration of the light path.

- Calculating Light Path

  Following the initialization, it is possible to calculate the light path in iterative manner. First of all, if the light path energy is too small it is possible to terminate (using Russian roulette).

  In case the path is not terminated, the hit point from the previous vertex and ray direction is calculated, used for computing the emissivity and color at this point. These are stored along with hit point as new vertex in the light path.

Further, the next ray direction is created (using importance sampling) and the next iteration of the light path computation can be computed.

Once the loop ends, e.g. the light path is terminated, all vertices on the light path are stored with all the data needed for performing a join with camera path.

- Calculating Camera path with Join

  Use primary ray origin and direction to calculate the hit point, thus calculating the first step of the camera path. On this hit point a path join with the light path has to be performed, the following case describes the full path join.

  For each vertex on the light path, determine visibility between the hit point and the light path vertex. For computing a visibility a ray is casted from the hit point to the light path vertex, in case it intersects a triangle prior to intersecting the light path vertex there is no visibility, otherwise there is. Each of the visible light path vertices adds contribution to the total energy of the camera path.

  A new ray direction is created (using importance sampling) using the previous hit point as new origin for the next iteration of the camera path. The camera path can be terminated using Russian roulette at the beginning of each iteration.

  Note that for perfectly reflective and refractive surfaces, the path join does not need to be computed as the weight for the energy contribution is equal to zero.

  The resulting value represents single sample per pixel of the computation.

4.2.4 Progressive Rendering

To allow for user interaction and real time experience the progressive rendering mode has been added into the implementation. This mode always calculates just a single sample per pixel and merges it with the data that are currently in the buffer (this buffer is drawn into the view port on the screen).

The merge works as following: Given the average of N samples stored in the output buffer as x, the new value x' is added as: $\frac{x \cdot N + x'}{N+1}$. Then the number of samples stored in the output buffer is incremented. While such computation is not as precise as holding a sum separately, there is no need for floating-point precision in the output buffer and also there is no need to perform the division by number of samples separately.

Upon camera movement or rotation, the output buffer is cleared to zero and the number of samples set to zero.

5 Results

The created implementation was tested on some of the common scenes in the computer graphics industry. For the purpose of comparison these scenes were rendered using other software, namely NVidia iRay and LuxRender.

While LuxRender is one of the largest renderers available, supporting subsurface materials, participating media and lots of other effects - it was one of the best candidates to compare, as implementation of LuxRays is GPU-based (built on OpenCL).

NVidia iRay does some of computations on GPU and is built on CUDA, yet it the used version does not support some of the effects the provided implementation does, like texturing for example.

5.1 Settings

All of the following scenes were rendered with the following settings:

- Unbiased rendering was used (the camera path length was unlimited, the maximum length of light path was defined to high value so that russian roulette most likely terminated it before reaching such length).
- Bounding Volume Hierarchy with Spatial Split was selected as an acceleration structure for all the scenes.
- Primary ray directions were generated in stochastic manner, so that resulting image is anti- aliased.

Further compared in terms of quality and speed. The quality comparisons are done in terms of root-mean-square error (further RMSE). Such comparison is used to measure the differences between estimated

values and actually observed values. In case of images the comparison is between the computed image and the ground truth[1].

5.2 Scenes

The following sections contain a brief description of each scene along with results in terms of quality and/or time. Note that for the Sponza atrium scene a comparison against NVidia iRay and LuxRender is provided.

5.2.1 Cornell Box

The Cornell box is a common scene for testing the quality of the global illumination algorithm. The scene is often composed of: a green wall, a red wall, white walls, a reflective object and a refractive object and also a light source.

Two resulting images after of computation on NVidia GeForce GT 720M (Kepler) and NVidia GeForce GTX 580 (Fermi) are provided, see 5.1. For simple scene like the Cornell box both of the GPUs generate perfect image almost instantly.

5.2.2 Modified Sponza

Sponza Atrium is a model originally created by Marko Dabrovic, further modified by Frank Meinl (his version is also known as Crytek Sponza). For the purpose of testing, further modifications were done. The following three tests compare:

1. The RMSE after 4 samples per pixel and 16 samples per pixel in computation. This scene is is modified, adds large cube-shaped

1. The ground truth is generated by unbiased algorithm that runs long enough to generate an image without any visible noise, such image is often considered as a correct solution, hence the ground truth.

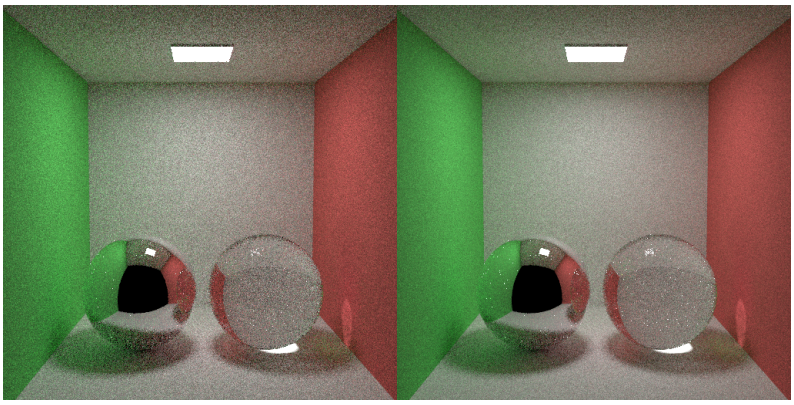

Figure 5.1: Both of the images are compared against the ground truth in terms of RMSE. On the left side an image generated by NVidia GeForce GT 720M in 0.1 second, having $RMSE = 0.0628$. On the right side an image generated by NVidia GeForce GTX 580 in 0.1 second, having $RMSE = 0.0456$.

light source and an object with glass material, see 5.2.

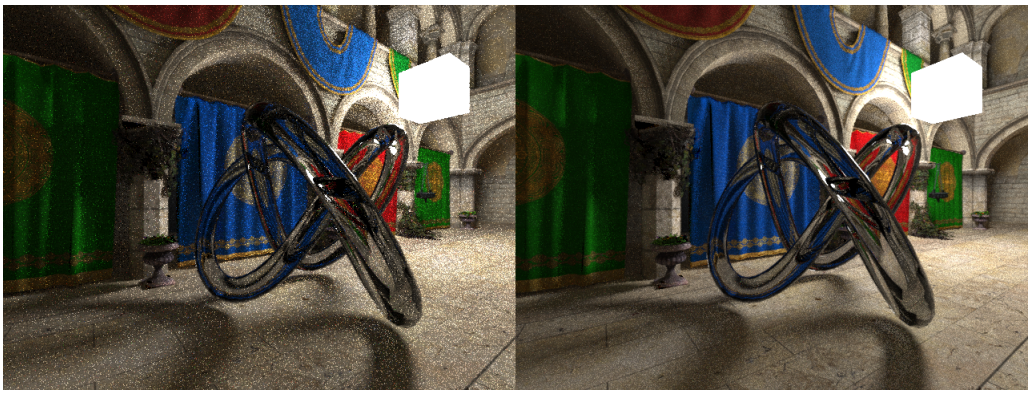

Figure 5.2: The left image has 4 sample per pixel with $RMSE = 0.0701$. The right image has 16 samples per pixel with $RMSE = 0.0095$.

2. The quality of progressive computation with multiple lights. The scene is modified with high intensity large source simulating sun and large low intensity light source with blue tint, simulating the

"skylight", see 5.3.

Figure 5.3: The following image shows part of Crytek Sponza with alpha tested plant inside stone object, the scene is lit using the sun and skylight. Image was taken after approximately a second of progressive computation on GeForce GeForce GT 720M.

3. The comparison between path tracing with explicit sampling and bi-directional path tracing. The scene contains a skylight simulating overcast weather. This is one of the hardest cases for path tracing with explicit sampling, bi-directional path tracing handles the case, see 5.4.

5.2.3 Sponza Atrium

The following scene is non-modified Crytek Sponza model by Marko Dabrovic and Frank Meinl. They are used for comparison of quality between the implemented renderer, Nvidia iRay and LuxRender. It is important to note, that each of these renderers have different set of features, hence the scene setup is not exactly the same for each image.

For the comparison with iRay similar conditions are used, iRay does not directly support lights defined by geometry so their scene uses

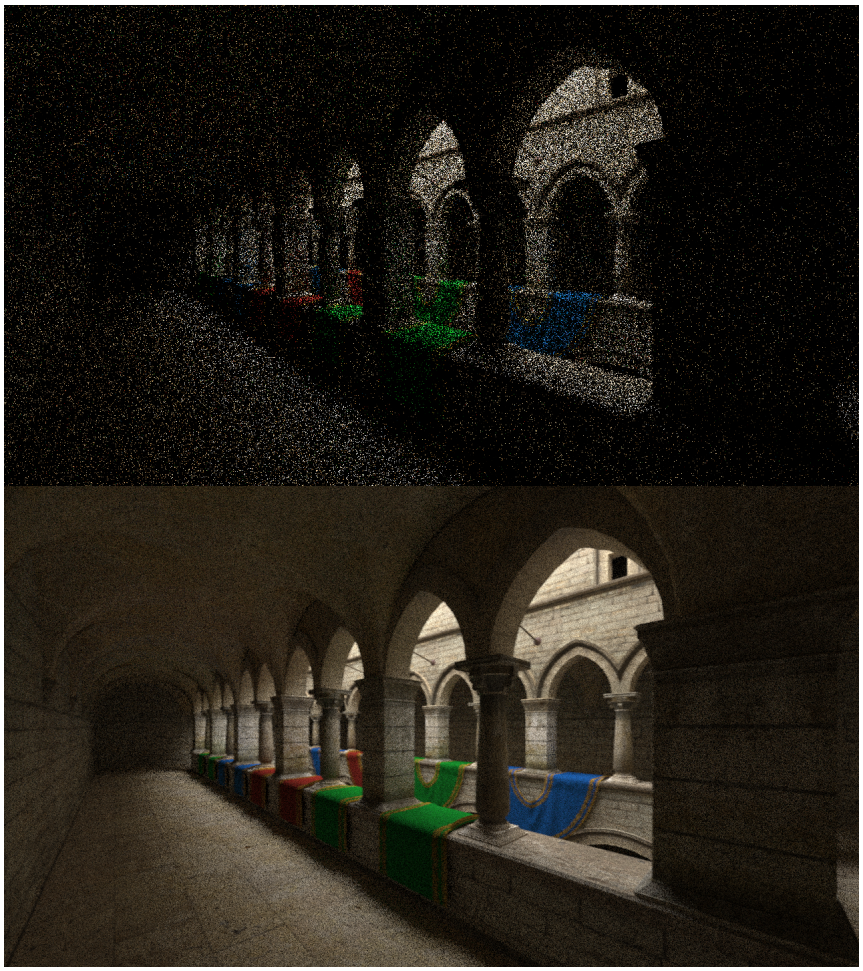

Figure 5.4: The top image was taken using standard path tracing with explicit sampling algorithm, the bottom image was taken using bi-directional path tracing algorithm. Both images took 15 seconds to render on NVidia GeForce GT 720M.

directional light, hence the shadow has a different shape. Also as both of the renderers use different post-processing, and thus the contrast in images is slightly different. For comparison, see 5.5.

The comparison between LuxRender and the implemented renderer is done also from similar view position. The setup for lighting is the

Figure 5.5: The comparison between iRay (left) and the implemented renderer (right). Both images took similar rendering time, approximately 60 seconds on NVidia GeForce GT 720M.

same, yet the camera position slightly differs.

Note that LuxRender results in slightly less noisy images as it does not use bi-directional path tracing algorithm, but the LuxRays algorithm (LuxRays is also unbiased rendering algorithm).

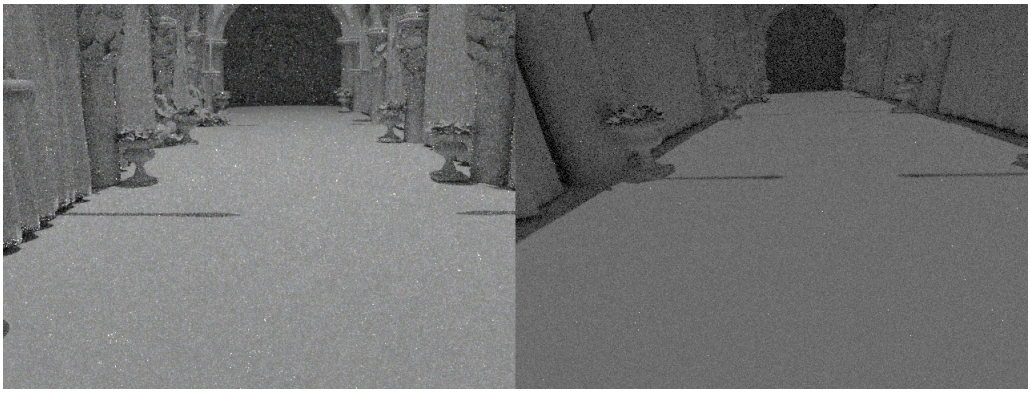

Figure 5.6: The comparison between LuxRender (left) and the implemented renderer (right). Both images took similar rendering time, approximately 15 seconds on NVidia GeForce GT 720M.

6 Conclusion

Bi-directional path tracing is one of the most robust rendering algorithms. The thesis proposed a way to implement parallel bi-directional path tracing and provides accompanying implementation which is a full bi-directional path tracing renderer. The implementation runs fully on the GPU, in terms of speed it achieves real time frame rates on high end graphics hardware, also note that even mobile hardware is able to achieve interactive frame rates.

It features advanced effects like lights specified using geometry, reflective and refractive materials and textures, thus allowing for rendering of real-world scenarios. The implementation is compared against standard path tracing solutions and production tools using GPU-based path tracing algorithms, the comparsion is done in terms of quality and speed.

The following short sections describe what could be done as possible future improvements of given implementation and therefore proposes possible future application on real world scenarios.

6.1 Real Time Rendering

One of the main challenges in ray tracing world is making a real time ray tracer, or ideally making a path tracer to run in real time with close to no noise.

The main goal of such software would be a simplification of rendering pipelines in current game engines and real time rendering software, which tends to hack the effects like reflections and global illumination, producing large and heavy rendering pipelines.

While there already are some implementations, like Brigade by J. Bikker [3], none of them is unbiased and physically based (these tend to heavily reduce the complexity of rendering pipelines).

6.2 Physically Based Rendering

A real challenge in bi-directional path tracing is physically based rendering, even though the created implementation allows for physically based rendering, it still supports only limited types of materials. The real challenge would be to implement fast, physically based renderer, supporting features including subsurface scattering and participation media, as these effects are still mostly hacked due to performance reasons.

Bibliography

[1] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on gpus. In *Proc. High-Performance Graphics 2009*, pages 145–149, 2009.

[2] Timo Aila, Samuli Laine, and Tero Karras. Understanding the efficiency of ray traversal on GPUs – Kepler and Fermi addendum. NVIDIA Technical Report NVR-2012-02, NVIDIA Corporation, June 2012.

[3] Jacco Bikker. Real-time ray tracing through the eyes of a game developer. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, RT '07, pages 1–10, Washington, DC, USA, 2007. IEEE Computer Society.

[4] Holger Dammertz, Johannes Hanika, and Alexander Keller. Shallow bounding volume hierarchies for fast simd ray tracing of incoherent rays. In *Proceedings of the Nineteenth Eurographics Conference on Rendering*, EGSR '08, pages 1225–1233, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association.

[5] Christer Ericson. *Real-Time Collision Detection (The Morgan Kaufmann Series in Interactive 3-D Technology) (The Morgan Kaufmann Series in Interactive 3D Technology)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[6] James T. Kajiya. The rendering equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86, pages 143–150, New York, NY, USA, 1986. ACM.

[7] Samuli Laine, Tero Karras, and Timo Aila. Megakernels consid-

ered harmful: Wavefront path tracing on gpus. In *Proc. High-Performance Graphics*, 2013.

[8] Matt Pharr and Greg Humphreys. *Physically Based Rendering, Second Edition: From Theory To Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2010.

[9] Christopher Shlick. An inexpensive brdf model for physically-based rendering. *Computer Graphics Forum 13*, 1994.

[10] Martin Stich, Heiko Friedrich, and Andreas Dietrich. Spatial splits in bounding volume hierarchies. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 7–13, New York, NY, USA, 2009. ACM.

[11] Eric Veach. *Robust Monte Carlo Methods for Light Transport Simulation*. PhD thesis, Stanford, CA, USA, 1998. AAI9837162.

[12] Craig M. Wittenbrink, Emmett Kilgariff, and Arjun Prabhu. Fermi gf100 gpu architecture. *IEEE Micro*, 31(2):50–59, March 2011.

[13] Sven Woop, Carsten Benthin, and Ingo Wald. Watertight ray/-triangle intersection. *Journal of Computer Graphics Techniques (JCGT)*, 2(1):65–82, June 2013.

List of Figures