Raytracing Render Switcher with Embree

Rion Sato[1,*] and *Michael* Cohen[1,**]

[1]Spatial Media Group, Computer Arts Lab.; University of Aizu; Tsuruga, Ikki-machi; Aizu-Wakamatsu, Fukushima 965-8580; Japan.

Abstract. We introduce a way of implementing physically-based renderers that can switch rendering methods with a raytracing library. Various physically-based rendering (PBR) methods can generate beautiful images that are close to human view of real world. However, comparison between corresponding pairs of pixels of image pairs generated by different rendering methods is necessary to verify whether the implementation correctly obeys mathematical models of PBR. For comparison, result images must be same scene, same resolution, from same camera angle. We explain fundamental theory of PBR first, and present overview of a library for PBR, Embree, developed by Intel, as a way of rendering-switchable implementation. Finally, we demonstrate computing result images by a renderer we developed. The renderer can switch rendering methods and be extended for other method implementations.

1 Introduction

1.1 Physically-Based Rendering

Photorealistic rendering, also known as physically-based rendering (PBR), presents beautiful images that approach view of real world by emulating natural process modeled by theories describing photon behavior. PBR uses simulation expressions modeling real world optical phenomena. Because of real world models' complexity, implementations of the models basically require heavy computation. A textbook that describes PBR [3] is useful to learn about PBR from fundamental theory to implementation.

1.1.1 Indirect Light

In real world, many photons are emitted from light sources, repeatedly reflecting off surfaces, before finally reaching our eyes. In computer graphics, many rendering algorithms omit the reflection aspect for lower computation cost. They consider reflection only from specular (mirror-like) surfaces and limit such reflection for other surfaces.

1.1.2 Illumination Models

In computer graphics there are two illumination models, local illumination (LI) and global illumination (GI). LI does not consider indirect light, which is reflection process with non-specular surfaces previously mentioned. GI models indirect light as well as direct light. See Figure 1 comparing LI and GI.

When implementing a rendering system, the renderer must delect which illumination model is suitable for the software. GI usually requires heavy computation time that can not be applied to realtime graphics processing such as videogames but can generate beautiful imagery offline, while LI is fast and applicable to computation for realtime processing and simple rendering. PBR is a category of GI-reproducible rendering methods. In addition, faked GI rendered by simplified methods are also called PBR.

1.1.3 Raytracing

For programming PBR, the raytracing method and its derivatives are generally used. They consider light directed and weighted linear projection, called rays. Many rays are cast from a virtual camera into a 3D scene, and simulation determines whether they hit any object. When hit, one can utilize the hit object's information such as normal and material for rendering calculation. Figure 2 depicts s a basic raytracing algorithm. There are a variety of calculation algorithms that are raytracing derivatives.

1.1.4 The Rendering Equation

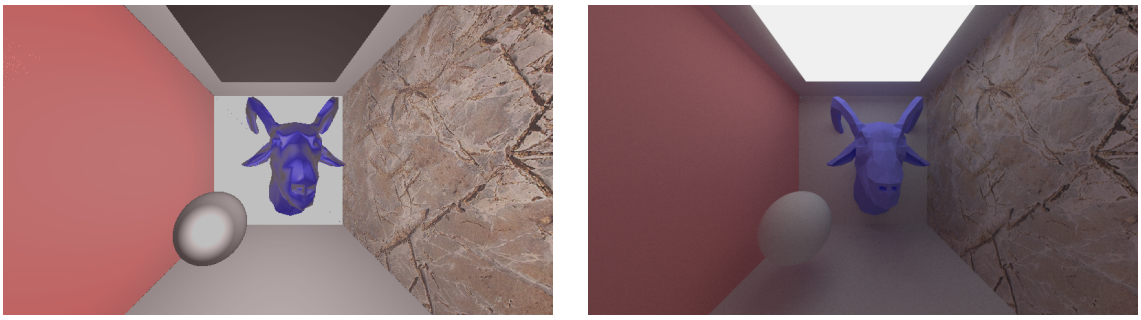
In 1986, an equation that generalizes GI rendering on computer was proposed by Kajiya [4]. The mathematical model adopted the idea that rays are infinitely reflected from any surface as in the real world, and expressed rendering calculation as a recursive integral form evaluated at each surface point:

$$L_o(\boldsymbol{x}, \vec{\omega_o}) = L_e(\boldsymbol{x}, \vec{\omega_o}) + \int_\Omega f_r(\boldsymbol{x}, \vec{\omega_i}, \vec{\omega_o}) L_i(\boldsymbol{x}, \vec{\omega_i}) |\vec{\omega_i} \cdot \vec{n_x}| d\vec{\omega_i} \tag{1}$$

where

- $\boldsymbol{x}$ is a surface point that a ray hit,
- $\vec{\omega_i}$ is incoming ray direction to $\boldsymbol{x}$ from previous point,

*e-mail: albus0sh@gmail.com
**e-mail: mcohen@u-aizu.ac.jp



(a) LI (b) GI

Figure 1: Rendered images by implementation of (a) LI and (b) GI. GI image is closer to real world realism. (a) is rendered by Phong shading algorithm with render time 0.08 s. (b) is rendered by pathtracing algorithm with render time 200.5 s. See §1.2 for the algorithms.

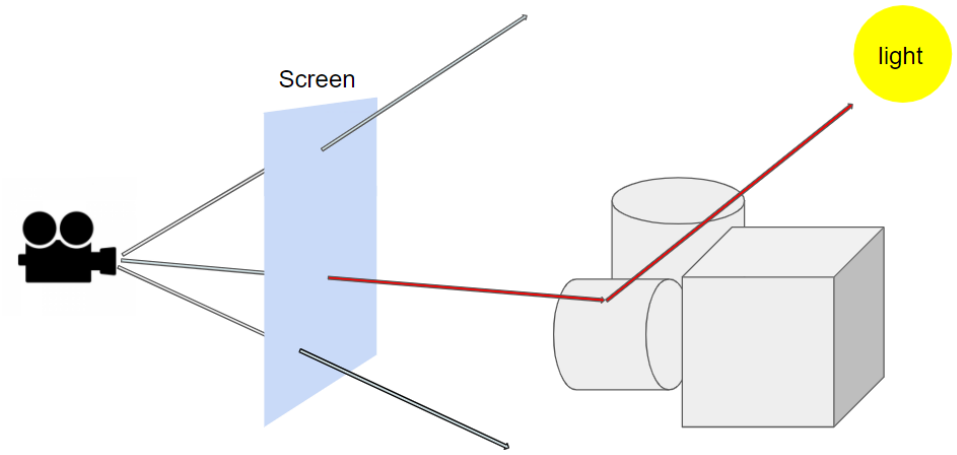


Figure 2: Visualization of basic raytracing algorithm. After object intersection, depending on derivative methods, various shading calculations are executed.

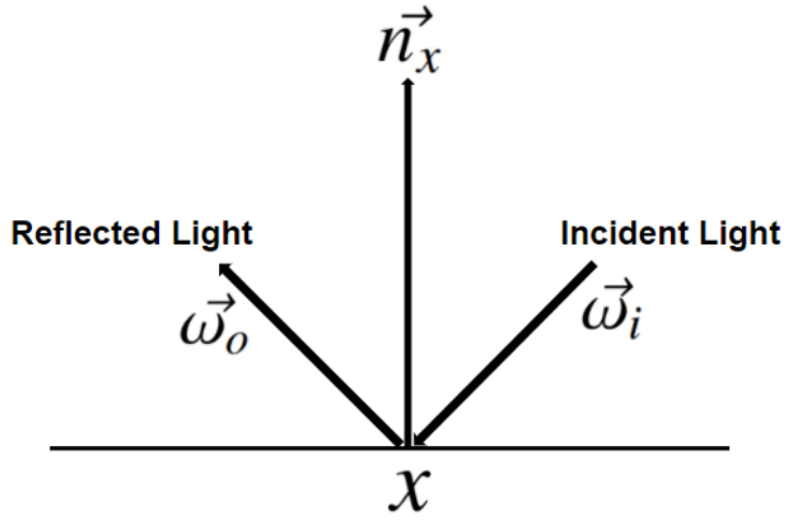


Figure 3: A surface point at which the rendering equation is evaluated. The equation is applied to each point that any ray intersects.

- $\vec{\omega_o}$ is outgoing ray direction from $\boldsymbol{x}$,
- $\vec{n_x}$ is normal to the surface at $\boldsymbol{x}$,
- $L_o(\boldsymbol{x}, \vec{\omega_o})$ is total light energy transported from $\boldsymbol{x}$ in the direction $\vec{\omega_o}$,
- $L_e(\boldsymbol{x}, \vec{\omega_o})$ is emitted light energy from $\boldsymbol{x}$ to $\vec{\omega_o}$,
- $L_i(\boldsymbol{x}, \vec{\omega_o})$ is transported light energy from the direction $\vec{\omega_i}$ to $\boldsymbol{x}$, and
- $f_r(\boldsymbol{x}, \vec{\omega_i}, \vec{\omega_o})$ is conductivity of light energy transported from $\vec{\omega_i}$ and reflected from $\boldsymbol{x}$ to $\vec{\omega_o}$. This is also known as bidirectional reflectance distribution function (BRDF). f_r describes $\boldsymbol{x}$'s material characteristic, such as diffuse(matte) or specular (glossy) surfaces. Details are in next section (§1.1.5).
- $|\vec{\omega_i} \cdot \vec{n_x}|$ is a factor to convert energy density of cross-sectional area around vector $\vec{\omega_i}$ to energy density of projected area that reaches point $\boldsymbol{x}$. (See Figure 4.)
- Ω is hemisphere that represents all ray-castable directions from $\boldsymbol{x}$.
- $d\vec{\omega_i}$ is infinitesimal solid angle.

This equation shows light energy amount that accumulates light energy incoming from all directions transports to a direction $\vec{\omega_o}$. Figure 3 depicts a point that is evaluated the rendering equation. A ray transports a number of photons, and the photons are described as a conductivity rate of emitted light energy from light sources. Iterative ray-casting and evaluating the equation at any surface point provides photorealistic images as the process is similar to light behavior of real world. Such recursive ray-casting from reflective points is called pathtracing, which is one of the raytracing derivatives. Other raytracing derivative methods also basically follow this equation. Most of them are accelerated versions of pathtracing obtained by modifying this equation, so approximately same value is expected.

For more detailed theory, Jensen's textbook [1] is helpful. This book explains the light transport theory and continues to a raytracing derivative, called photon mapping. ([2] is Japanese translation of first edition of [1].) [3, §5.6.1 and §14] also contains a good explanation.

1.1.5 Bidirectional Reflectance Distribution Function (BRDF)

Characteristics of any surface, also known in CG as "material," are expressed in PBR as weighting functions called Bidirectional Reflectance Distribution Functions (BRDF). A BRDF takes ray-intersected position, incoming ray direction, and outgoing ray direction to calculate how much energy transported to a point x is reflected to particular direction. For example, non-glossy (matte) surfaces reflect energy in all directions uniformly, while ideal specular (mirror-like) surfaces reflect maximal weight in a particular direction.

1.2 PBR methods

A variety of PBR methods have been researching to realize photorealism. In this section, we introduce an overview of well-known algorithms in graphics community.

1.2.1 Phong Illumination Model

Phong's illumination model [6] is a very approximate rendering model (LI), mainly used for realtime graphics, but not photorealism. Its shading calculation is modeled as an accumulation (addition) of following:

- Ambient light: light power added to all 3D objects' surfaces. This is often expressed as a constant.
- Diffuse reflection: contribution of non-glossy (diffuse) reflected light energy.
- Specular reflection: contribution of glossy surfaces.

The two reflection terms are easily calculated from relationship between incident ray direction and surface normal, so this is often used for graphics programming tutorials.

1.2.2 Pathtracing

In 1986, pathtracing [4] was proposed. Pathtracing is the simplest algorithm that naturally follows the rendering equation. Because of its simplicity, it is often used as a "ground truth" algorithm to compare other algorithms' correctness (as elaborated below in §1.3). Other raytracing derivatives are basically extensions of this algorithm. The recursive form equation is evaluated through the following process:

1. A ray is cast from a point. Casting direction is randomly determined, other than first ray from camera.
2. Perform intersection test. If the ray hit something, continue to next step. Otherwise, terminate this process (as "failure") ,with no light energy transported.
3. Evaluate the rendering equation at the intersected point.
4. Repeat this process (evaluate the equation recursively) until particular condition is fulfilled. In general, stochastic "Russian roulette" is used.

1.2.3 Pathtraing with Next Event Estimation (NEE)

Pathtracing is less practical than recent raytracing methods because of random direction sampling. Next Event Estimation (NEE) is an improvement. At each intersection test, check whether each intersected point can connect to see light with no obstacles. If viewable, accumulate additional contribution of light energy.

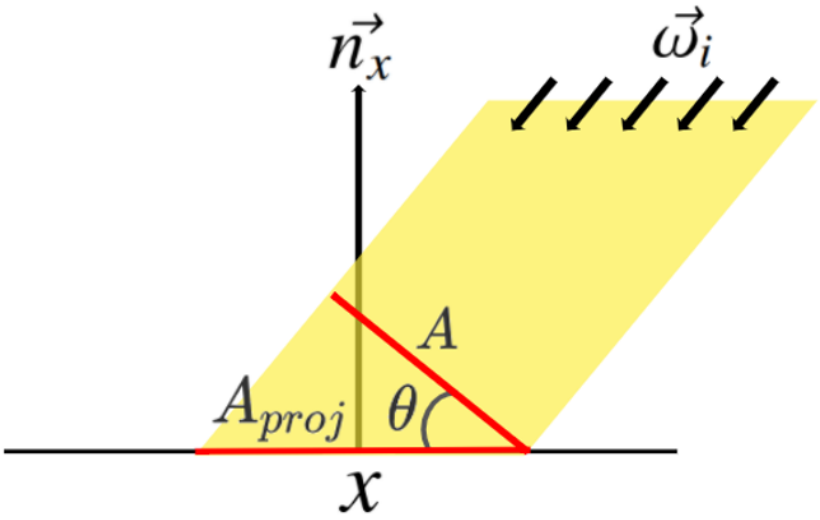


Figure 4: To consider the number of photons that go through infinitesimal area (represented by infinitesimal solid angle $d\vec{\omega_i}$) on a point $\boldsymbol{x}$ on a surface, $\cos\theta$, which equals dot product alignment between normalized vectors $\vec{\omega_i}$ and $\vec{n_x}$, is required for area expandability because of diagonal incidence. Original area is A, while actual area that is illuminated by a ray is A_{proj}, so $\cos\theta = |\vec{\omega_i} \cdot \vec{n_x}|$ scales the contribution.

1.2.4 Bidirectional Pathtracing

Bidirectional Pathtracing (BDPT) is a derivative of NEE. (For details, see [3, §16.3].) BDPT accumulates history of intersected points and their contributions. After finishing normal pathtracing for a path, a combination of points that have highest contributions from the history are used to evaluate the rendering equation. For this process, more efficient rendering convergence speed can be realized.

1.2.5 Photon Mapping (PM)

Photon mapping (PM) is a somewhat tricky refinement of pathtracing. In PM, process is divided into two steps, as seen in Figure,5. (For details, refer to [1] or its Japanese translation [2].)

1. From points on surfaces of light source objects, the algorithm casts photons that have light energy. Next, repeat photon casting on intersected surfaces stochastically, like ray casting of general pathtracing. When each photon is stopped, its position is saved to a data structure, called photon map.

2. Raytrace to the photon map, from a virtual camera, through virtual screen to be rendered. Count number of photons around first intersected point and calculate light energy, based on the number of the "neighborhood" photons.

1.2.6 Metropolis Light Transport

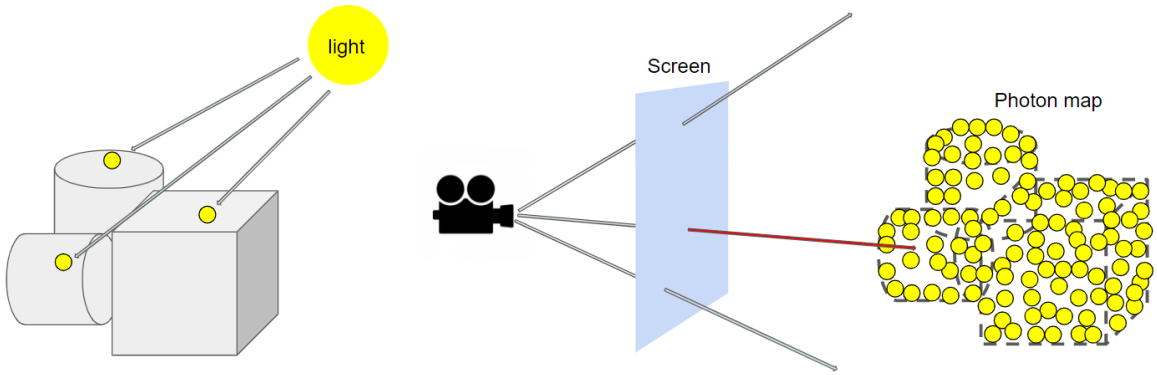
Metropolis Light Transport (MLT) is a pathtracing derivative developed with the idea that reusing light-reachable paths has high contribution to light energy transporting ([5] is the published thesis, and also [3, §16.4] provides plain explanation). When a path that successfully reached light sources are found by general pathtracing, from next time pathtracing, the reached path is slightly changed its directions as different path to light, and reused to compute light energy.

1.3 Rendering Correctness

When implementing a PBR method, we have to verify the renderer is correct. To confirm, comparison between a "ground truth" reference image and an image generated by the renderer is required. Subtraction of the two images' corresponding pixel values visualizes such difference. If the renderer is correctly implemented, the difference of corresponding pixel values of two images vanishes (approaches 0). (Completely absent noise is impossible because every PBR implementation is based on stochastic process, so random variations are unavoidable.) A way of preparing the reference image is implementing renderer of simplest PBR method. In many PBR renderers, pathtracing algorithm is used because of its simplicity. Figure 6 shows an instance of subtraction between Phong shading and pathtracing.

1.4 Goal

As explained in §1.3, taking difference between two images of same scene is important for validating renderings. This project aims at showing a way of comparison among PBR methods. We present how to develop method-switchable renderer using a raytracing library, Embree by Intel [7]. Embree is an open source library developed at [8].



(a) Step 1: Photon casting (b) Step 2: Photon counting near the intersection point

Figure 5: Steps of photon mapping. Many photons are cast from lights to construct a photon map (a). Then, the photon map is raytraced from camera through a virtual screen (b).

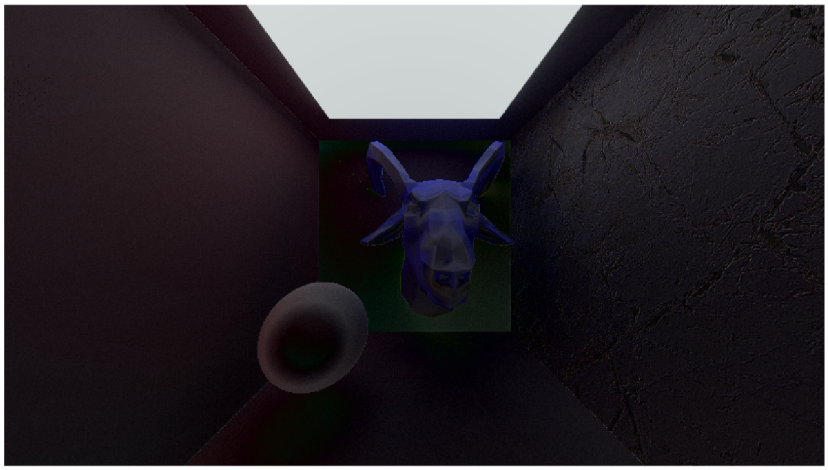


Figure 6: Difference between images in Figure 1. Red color from indirect lighting can be seen on leftside surface of the sphere. Rendering difference of shadow is also visualized under the goat sculpture. Easiness of seeing goat means great difference in light transport calculation between Phong shading and pathtracing.

2 Embree Raytracing

In this section, we present Embree features. For more programming details, please see a tutorial written by first author for actual C++ programming with Embree [10].

2.1 About Embree

Embree is a C99 library that enables fast raytracing on CPU. It is optimized for Intel processors and supports other CPUs that can invoke SIMD intrinsics: SSE, AVX, AVX2, and AVX-512. One can implement raytracing renderer by calling Embree APIs. The kernel of Embree does optimized invoking of the SIMD intrinsics.

2.2 Embree API

In Embree, there are special objects that have particular roles of the API. Programmers utilize them in particular order to operate Embree kernel that executes necessary processes of raytracing. See Figure 7.

2.2.1 Device Initialization

First of all, a **device** of Embree objects is required to operate Embree kernel. The instantiated device is used to generate other Embree objects. Device can be configured by specific string in some ways. For example, programmers can set number of CPU threads that Embree kernel can occupies and type of CPU's SIMD instruction architecture sets such as SSE, AVX. For more detail of configuration, see `rtcNewDevice` of the official Embree reference.

2.2.2 Scene Initialization

A scene of Embree objects contains and manages "3D model instances" in virtual space. Scene will be used for a

Figure 7: Order of Embree API call.

special data structure for raytracing, which is an internally-constructed bounding volume hierarchy (See [3, §4.3]), which is grouped 3D-boxes, to compute ray-object intersection.

2.2.3 Geometry Initialization

Geometry is an Embree object corresponding to a "3D model instance". Geometry's data buffers in memory is allocated when constructing its instance. Programmer can specify its attributes and memory layout by Embree functions for geometry instantiation. Multiple geometries will be registered to a scene object to render.

2.2.4 Casting a Ray

Embree's `rtcIntersect1` and equivalent functions executes actual raytracing process. A scene and instances to save geometric information will be handed as parameters, and Embree kernel queries whether a ray hit to some 3D model instances, to the scene.

2.3 Raytracing Rendering with Screen

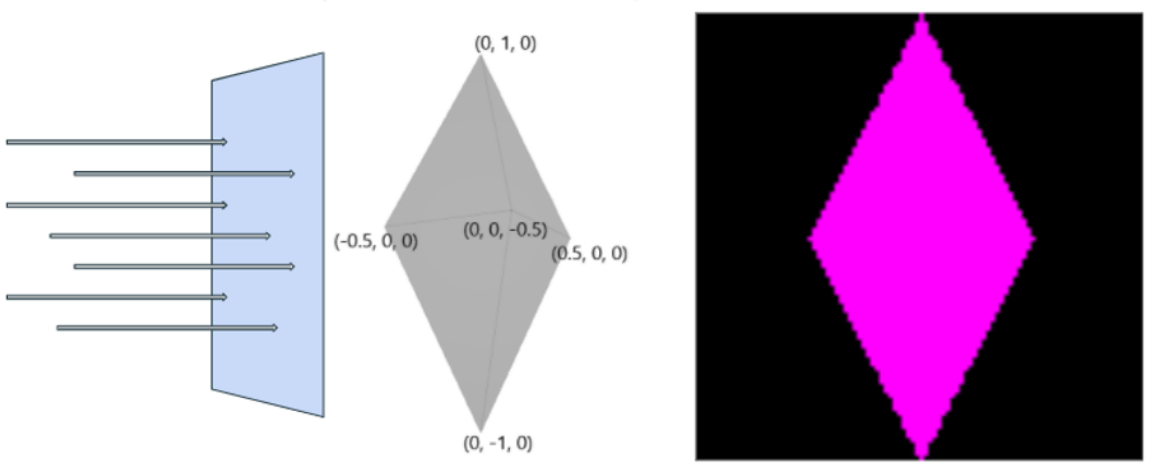
Considering a rendering screen as image buffer in virtual 3D space can visualize shape of 3D model instances. Following pseudo-code shows a simplest implementation of a raytracing renderer, which just visualizes scene silhouette. See Figure 8.

```
1  image = 0-initialized image
2  for each row of image {
3    for each pixel of a row {
4      origin = compute raytracing start point
5      dir = compute first raytracing direction
6      if(ray(origin, dir).intersects(scene)) {
7        image[row][pixel] = MAGENTA
8      }
9    }
10 }
```

When implementing actual PBR methods, the `MAGENTA` in line 7 is replaced by the color intensity computed from the rendering equation (§1.1.4).

2.4 Renderer Implementation

To switch rendering methods, the logic part of rendering should be brought out as a function, so one can switch rendering methods by conditional branching or otherwise polymorphism. We do not refer to polymorphism and implementation of specific rendering methods in this research. ([1] and [2] include explanation of primitive raytracing algorithms, so they are helpful to implement.) Rendering of a pixel can be considered independent in raytracing, so inside of the double loop of §2.3 is same as that in following pseudo-code:



(a) Virtual screen and 3D-model.

(b) Rendering result.

Figure 8: Simplest raytracing that visualizes silhouette of a 3D model. Rays are screen-orthogonally cast as going through each pixel.

```
renderCond = user input
image = 0-initialized image
for each row of image {
  for each pixel of a row {
    origin = compute raytracing start point
    dir = compute first raytracing direction
    switch renderCond {
        case A:
          image[row][pixel] = RenderMethodA(
              scene, origin, dir)
        case B:
          image[row][pixel] = RenderMethodB(
              scene, origin, dir)
        case C:
          image[row][pixel] = RenderMethodC(
              scene, origin, dir)
    }
  }
}
```

3 Result

We implemented a renderer by Embree. Result images are indicated in Figure 1. Visit our GitHub repository [9] if you are interested in implemented C++ programs.

4 Discussion

We could obtain two images of same scene, same resolution, from same angle, by different methods. As mentioned in §1.3, same condition of image is necessary to apply subtraction. Figure 6 shows actual subtraction of the two images. We now could know where areas are incorrectly rendered in the scene. When debugging rendering program, knowing strangely-rendered locations is important to determine what the program problems are.

5 Conclusion

We have introduced background of PBR and presented overview of raytracing with a library, Embree. PBR simulates photon behavior of real world by mathematical models, and needs heavy computation that can not be applied to realtime graphics processing because of their complexity. Each PBR method is researched and should converge same rendering result in order to obey correctness of mathematical models. To verify implementation of each PBR method converges same rendering result, each-pixel subtraction between two images rendered by two different methods are important.

A library Embree enables your raytracing program faster by internal implementation with CPU-intrinsic SIMD functions. Embree has a feature of encapsulation of a scene that can contain instances of 3D models, so rendering to same scene by various methods can be realized.

We have implemented a method-switchable renderer with Embree. The renderer can show images that are same position of view, same scene, same resolution, for subtraction of two images. As future work, more implementations of other PBR methods and GPU acceleration such as compute shaders, GPGPU can be added to the renderer.

References

[1] Henrik Wann Jensen, *Realistic Image Synthesis using Photon Mapping, 2nd edition* (AK Peters, America, 2005), ISBN 1568811407

[2] Henrik Wann Jensen, translated by Takeshi Naemura, *Realistic Image Synthesis using Photon Mapping, Japanese edition* (Ohmsha, Japan, 2001), ISBN 4274079503

[3] Matt Pharr, Wenzel Jakob and Greg Humphreys, *Physically Based Rendering From Theory to Implementation Third Edition* (Morgan Kaufmann, America, 2016), e-book ISBN 9780128007099, hardcover ISBN 9780128006450

[4] James T. Kajiya, "The Rendering Equation". SIGGRAPH '86: Proc. 13th Annual Conf. on Computer

Graphics and Interactive Techniques, pages 143–150 (August, 1986), DOI 10.1145/15922.15902

[5] Eric Veach and Leonidas J. Guibas, "Metropolis Light Transport". SIGGRAPH '97: Proc. 24th Annual Conf. on Computer Graphics and Interactive Techniques, pages 65-76 (August, 1997), DOI 10.1145/258734.258775

[6] Bui Tuong Phong, "Illumination for Computer Generated Pictures". Communications of the ACM, pages 311-317 (June, 1975), DOI 10.1145/360825.360839

[7] Embree official cite. https://www.embree.org

[8] Embree GitHub repository. https://github.com/embree/embree

[9] CGRS GitHub repository. https://github.com/Nao-Shirotsu/Embree-CGES

[10] Tutorial of raytracing programming with Embree. https://github.com/Nao-Shirotsu/Embree-CGES/blob/master/embree_intro_en.md