Project Proposal: Geocold Ray Tracer (Differentiable?) in C++

Authors: Prakash Chaulagain, Nishar Arjyal, and Pramish Paudel
Roll Numbers: 076BCT045, 076BCT042, 076BCT047

Submitted to the Department of Electronics and Computer Engineering in Partial Fulfillment of the Requirements of the 3rd Year **Computer Graphics** Course at
Pulchowk Campus
IOE, Tribhuwan University

June 29, 2022

Accepted by: .....

Mr. Basanta Joshi
Assistant Professor at
The Department of Electronics and Computer Engineering

Date of Submission: June 29, 2022
Expected Date of Completion: July, 2022

Geocold

by Prakash Chaulagain, Nishar Arjyal, and Pramish Paudel

Submitted to the Department of Electronics and Computer Engineering
on June 29, 2022
in Partial Fulfillment of the Requirements for the 3rd Year Computer Graphics Course in Computer Engineering

Abstract

Ray tracing has a rich history in the history of computing and computer graphics. With this manuscript, we propose to build an offline ray tracing software using the Vulkan graphics/compute library in C++. Our renderer is supposed to work generically, as in take as input any file containing geometric data, perform a mesh render pass in order to render a mesh of the described scene and then perform ray tracing with a separate pass. In this paper, we cover the mathematical principles that we follow as we build our ray tracing software.

Table Of Contents

1 Acknowledgement

Our project idea is the product of excellent supervision of all of our instructors, most notably our lecturer Mr. Basanta Joshi. We could not have been able to develop interest in computer graphics as a field of study without the constant inspiration provided to us by all of our lecturers and lab instructors and assistants. Some of the credit should also go to the college administration for their efforts in the smooth functioning of all of our classes and labs safely and securely despite the unprecedented times of the pandemic.

> *"He who loves practice without theory is like the sailor who boards ship without a rudder and compass and never knows where he may cast."*
> —Leonardo Da Vinci, 1452-1519

2 Objectives

- To understand the graphics pipeline.
- To become familiar with modern GPU architectures.
- To become familiar with GPU programming models and GPU computing (massively parallel computing).
- To understand and uncover existing ray tracing techniques.
- To gain a degree of familiarity with common graphics and GPU compute APIs, and understand their abstraction mechanisms.

3 Introduction

Rendering is the task of taking a scene composed of many geometric objects arranged in 3D space and computing a 2D image that shows the object as viewed from a particular viewpoint. The goal of our project Geocold is to create a photorealistic renderer by implementing a .obj file loader which then creates a mesh of our scene, then finally we implement a ray tracer which will correctly color every object in the scene. Over the next few sections, we will try and set the mathematical basis/principles used in our project and the API that we have attempted to design based on those principles.

4 Theory

In order to explain the way our ray tracer works, we need to set the ground with some basic mathematical terms and notations. This will not only give us a more technical basis for coming up with a good, mathematically correct API, but also help someone using the same piece of code to connect the pieces together. The next few of definitions are based on Farin([2]) and Goldman([3]).

Definition 1 (Affine Combination in 3D)**.** *An affine combination of vectors* $\boldsymbol{x_1}, \boldsymbol{x_2}, \boldsymbol{x_3}$, $\boldsymbol{x_i} \in \mathbb{R}^3$ *is the linear combination* $\sum_{i=1}^{3} \lambda_j \boldsymbol{x_i}$ *such that* $\sum_{j=1}^{3} \lambda_j = 1$

Definition 2 (Affine Map)**.** $\Phi : \mathbb{R}^3 \to \mathbb{R}^3$ *is an affine map if it leaves affine combinations invariant. That is, if*

$$\boldsymbol{a} = \sum_{j=1}^{3} \lambda_j \boldsymbol{x_j}, \sum_{j=1}^{3} \lambda_j = 1, \boldsymbol{a}, x_j \in \mathbb{R}^3$$

then,

$$\Phi(\boldsymbol{a}) = \sum_{j=1}^{3} \lambda_j \Phi(x_j); \Phi(x_j), \Phi(\boldsymbol{a}) \in \mathbb{R}^3. \tag{1}$$

The equation(1) specifies the correct way of weighing the points x_j such that their weighted average is the point $\boldsymbol{a}$.

In a given coordinate system, a point $\boldsymbol{x}$ is represented in the form of a coordinate triple, which we also denote by $\boldsymbol{x}$. An affine map now takes the form

$$\Phi(\boldsymbol{x}) = A\boldsymbol{x} + \boldsymbol{v} \tag{2}$$

The proof that equation(2) is affine is in Farin[2].

4.1 On Points and Vectors

Goldman (see [3]) describes how points and vectors are different and ought to be treated differently. Their work also describes the way of dealing with their differences.

Points have positions but not direction or length, while vectors have a direction and a length. Not all operations that can be applied for vectors can be applied for points. Points cannot be added, however addition like operations such as an affine combination (also called *barycentric combination*). Although we will not go into a comprehensive list of operations that can be applied for points and for vectors, we will list (table:1) out some of the common operations that are commonly implemented in any legitimate ray tracing API (like PBRT[8] and the ones mentioned in Nvidia's Ray Tracing Gems[4]). We denote points with capital letters like P or Q, and we denote vectors with small boldface letters like $\boldsymbol{u}$ or $\boldsymbol{v}$.

To come up with a decent and mathematically correct ray tracer, it is essential that we use the C++ type system to deal with these inherent differences between mathematical constructs.

Operation	Legal	Undefined
Addition	$\boldsymbol{u}$+$\boldsymbol{v}$(v), P+$\boldsymbol{u}$(p)	P+Q
Subtraction	$\boldsymbol{u}$-$\boldsymbol{v}$(v), P-$\boldsymbol{u}$	$\boldsymbol{u} - \boldsymbol{P}$
Scalar Multiplication	c.$\boldsymbol{v}$(v)	c.P
Dot Product	$\boldsymbol{u}.\boldsymbol{v}$(scalar)	P.$\boldsymbol{u}$, P.Q
Cross Product	$\boldsymbol{u} \times \boldsymbol{v}$(v)	P $\times$Q, P $\times\boldsymbol{u}$

Table 1: Common operations defined for points and vectors. (v in the braces represents vector and p represents point)

Listing 1: Sample code for Point3 and Vec3 types

```
template<typename T>
struct Vec3; //forward declaration

template <typename T>
struct Point3 {
   T x_;
   T y_;
   T z_;
   Point3(T x, T y, T z) noexcept
      :x_{x},
      y_{y},
      z_{z} {}

   Point3 operator+(const Vec3<T>& vec) noexcept {
      return Point3<T>{
         x_ + vec.x_,
         y_ + vec.y_,
         z_ + vec.z_
      };
   } //this implements Point3 + Vec3

   Vec3<T> operator-(const Point3& rhs) noexcept {
      return Vec3<T>{
         x_ - rhs.x_,
         y_ - rhs.y_,
         z_ - rhs.z_
      };
   } // Point3 - Point3 -> Vec3
```

```
        Point3<T> operator-(const Vec3<T>& vec) noexcept {
            return Point3<T>{
                x_ - vec.x_,
                y_ - vec.y_,
                z_ - vec.z_
            };
        }
    };

    template <typename T>
    struct Vec3<T> {
        T x_;
        T y_;
        T z_;

        Vec3(T x, T y, T z) noexcept
            :x_{x}
            ,y_{y}
            ,z_{z} {}

        Vec3<T> operator-() const {
            return Vec3<T>{
                -x_,
                -y_,
                -z_
            };
        }

        Vec3<T> operator*(T scalar) {
            return Vec3<T>{
                x*scalar,
                y*scalar,
                z*scalar
            };
        }

//other trivial operations like
//vector addition,
//subtraction, dot product and cross
//product are also implemented.
//Vec3 - Point3 is not implemented
```

```
//as it is an invalid operation.
   };
```

4.2 Other Primitives

Definition 3 (Normal)**.** *A surface normal is a vector perpendicular to a surface at a particular position.*

It is necessary to have a separate type for normals as they are treated differently to vectors (normals take object properties into account). Advanced ray tracers (like [8]) also take into account the various partial derivatives of the surface normal with respect to the local parametrization of the curve. In ray tracing, it is essential to know the normal to the surface at which the ray-triangle intersection test is satisfied so as to compute the direction in which the ray will be reflected next.

Definition 4 (Ray)**.** *We represent a ray as a parametric line in Geocold. If* $\boldsymbol{a}$ *and* $\boldsymbol{b}$ *are two points such that the ray starts at* $\boldsymbol{a}$ *then, the line between* $\boldsymbol{a}$ *and* $\boldsymbol{b}$ *is given by the parametric equation:*

$$\boldsymbol{r}(t) = (1-t)\boldsymbol{a} + t\boldsymbol{b}, t \in [0,1] \tag{3}$$

This is obviously an affine combination of the points $\boldsymbol{a}$ and $\boldsymbol{b}$. However, on slight modification, we can implement a ray as a linear combination of a point and a vector as such:

$$\boldsymbol{r}(t) = \boldsymbol{a} + t(\boldsymbol{b} - \boldsymbol{a}) \tag{4}$$

But, we need the parameter t to be in a set interval so that we won't have to track the ray farther than we need to (upto when the ray passes the ray-triangle intersection test). The ray can also not go behind the camera or eye. So, we need fields *tmin* and *tmax* for our struct.

```
struct Ray {
   Point3<float> a;
   float tmin;
   Vec3<T> direction; //b-a
   float tmax;
}; //ray is left unparametrized
```

Since most computer graphics projects define common camera models, we won't go to depth on the camera model for ray tracing in this paper.

We will finally end this section with a few words on radiosity and light model.

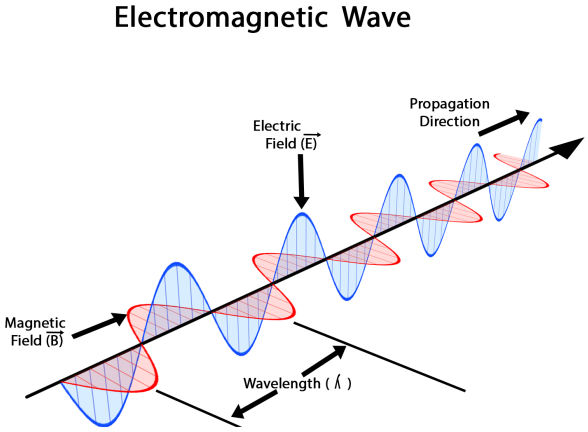

Figure 1: Light as an electromagnetic wave.

4.3 Raidosity and Light Models

Light is an electromagnetic wave that propagates through space. A convincing computer model for light would incorporate into it the direction of propagation, speed, wave-lenth, amplitude, polarization and so on. Although not all of these parameters are incorporated in standard ray tracers, ours will be a lightweight one in which we shall model light without any polarizability. The vector form for the law of reflection of light is as follows:

$$\boldsymbol{\omega}_r = \boldsymbol{\omega}_i - 2\boldsymbol{n}_p(\boldsymbol{\omega}_i.\boldsymbol{n}_p)$$

4.3.1 Whitted Ray Tracing

Turner Whitted in his paper 'An Improved Illumination Model For Shaded Display'[10] highlighted the use of geometric optics in rendering. He improved upon the Phong illumination model[9] by replacing specular component in his model by the intensity S of the light coming from the direction of specular reflection. His model also accounted for transmitted light intensity by adding T term weighted by k_t coefficient of transmitted light contribution.

$$I = I_a + k_d \sum_{i}^{l} L_{i,d}(\boldsymbol{n}.\boldsymbol{\omega}_i) + k_s S + k_t T \tag{5}$$

where:
I_a = intensity of ambient component
k_d and k_s = diffues and specular coefficients

$L_{i,d}$ = diffuse intensity of ith light source
$\boldsymbol{n}$ = equivalent to reflection sharpness in Phong[9], handled by adjusting random perturbations in $\boldsymbol{n}$ vector
l = number of light sources

S and T intensities depend on previous light hits and hence, must be calculated recursively.

4.4 Measuring Light

Our ray tracer however, should be able to handle the effect of the wavelength of light in the ways it behaves after striking a surface/object in the scene. Our ray tracer will disregard the spectrums and only deal with RGB. First, let's define some terms from radiometry. A rigorous discussion of the following theory is available in Lafortune[7] and Wojciech Jarosz's Ph.D. Dissertation[5] on the same topic also does an excellent job at making these ideas clear.

Definition 5 (Radiant Energy)**.** *Electromagnetic radiations carry energy through space. The energy carried by a single photon emitted by a source of illumination is given by*

$$Q = \frac{hc}{\lambda}$$

Definition 6 (Radiant Flux)**.** *Radiant Flux or power is defined as the total amount of energy passing through a surface or region of space per unit time.*

$$\Phi = \frac{dQ}{dt}$$

Definition 7 (Radiant Flux Density)**.** *Radiant Flux Density is the radiant flux per unit area at a point on a surface, where the surface could be real or imaginary.*

Definition 8 (Irradiance E)**.** *The radiant flux arriving at a surface per unit area is called irradiance.*

$$E = \frac{d\Phi}{dA}$$

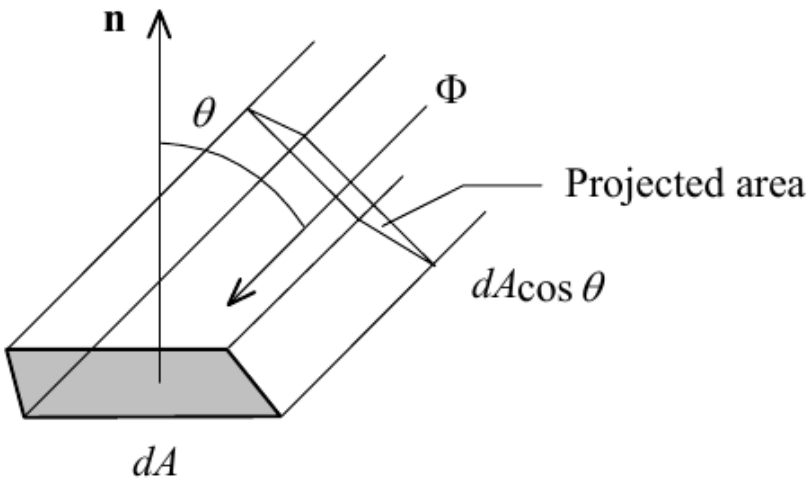

Figure 2: Ray of Light Intersecting a surface

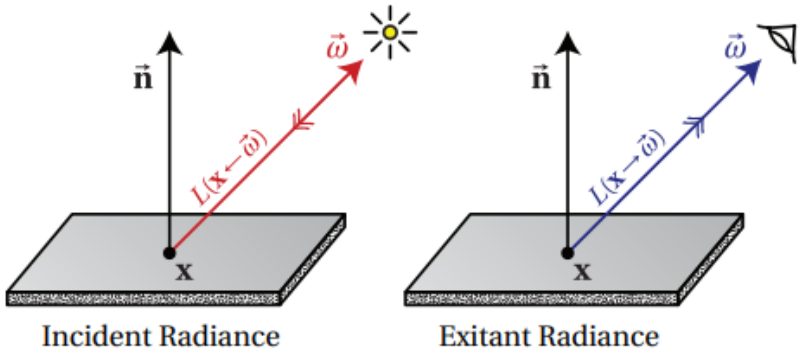

Figure 3: Incident and Exitant Radiance

Definition 9 (Radiant Exitance M)**.** *The radiant flux leaving from a surface per unit area is called radiant exitance.*

$$M = \frac{d\Phi}{dA}$$

Definition 10 (Radiance)**.** *Imagine an elemental cone to represent a ray that intersects a surface making an angle θ with the surface normal as in figure 2. The projected area of the ray-surface intersection area dA is $dAcos\theta$. The radiance is the radiant flux density per unit elemental cone or per unit solid angle $d\omega$.*

$$L = \frac{d^2\Phi}{dAcos\theta d\omega} = \frac{d^2\Phi(\boldsymbol{p}, \boldsymbol{\omega})}{(\boldsymbol{n}_p.\boldsymbol{\omega})d\boldsymbol{\omega}dA(\boldsymbol{p})} \tag{6}$$

We can also integrate equation (6) over the hemisphere of directions Ω and area dA to obtain

$$\Phi = \int_A \int_\Omega L(\boldsymbol{p} \rightarrow \boldsymbol{\omega})(\boldsymbol{n}_p.\boldsymbol{\omega})d\boldsymbol{\omega}dA(\boldsymbol{p})$$

We can also compute the irradiance and radiant exitance using :

$$E(\boldsymbol{p}) = \int_\Omega L(\boldsymbol{p} \leftarrow \boldsymbol{\omega})(\boldsymbol{n}_p.\boldsymbol{\omega})d\boldsymbol{\omega} \tag{7}$$

$$M(\boldsymbol{p}) = \int_\Omega L(\boldsymbol{p} \rightarrow \boldsymbol{\omega})(\boldsymbol{n}_p.\boldsymbol{\omega})d\boldsymbol{\omega} \tag{8}$$

Although in the case of surfaces in general,

$$L(\boldsymbol{p} \rightarrow \boldsymbol{\omega}) \neq L(\boldsymbol{p} \leftarrow \boldsymbol{\omega})$$

In vacuum where photons can propagate unobstructed, all the radiance incident at a point from a direction $\boldsymbol{\omega}$ will continue on in the form of exitant radiance in the direction $-\boldsymbol{\omega}$. Because of this, we can form a simple relation between exitant and incident radiance at a point. To accomplish, we can come up with a ray casting function $\boldsymbol{r(p, \omega)} = \boldsymbol{p}$' which returns $\boldsymbol{p}$', the point on the closest surface from $\boldsymbol{p}$ in the direction $\boldsymbol{\omega}$. Since radiance stays constant in vacuum along a straight line, we can say that the incidence radiance at a point $\boldsymbol{p}$ from direction $\boldsymbol{\omega}$ is equal to the outgoing radiance from the closest visible points in that direction. That is

$$L(\boldsymbol{p} \leftarrow \boldsymbol{\omega}) = L(\boldsymbol{p}' \rightarrow -\boldsymbol{\omega}) \tag{9}$$

Although our ray tracer will not incorporate the effect of participating medium on the rays, we will have to take the material properties into account as even the most minimal of ray tracers ought to do.

4.5 Interaction of Light with Surfaces

4.5.1 The Bidirectional Reflectance Distribution Function

If $\boldsymbol{\omega}_i$ be the incoming direction, and $\boldsymbol{\omega}_0$ be the outgoing direction, then the BRDF(f_r) gives a measure for the amount of radiance impinging at a point $\boldsymbol{p}$ along a direction $\boldsymbol{\omega}_i$ that is reflected along $\boldsymbol{\omega}_0$. Formally,

$$f_r(\boldsymbol{\omega}_i, \boldsymbol{p}, \boldsymbol{\omega}_o) = \frac{dL_o(\boldsymbol{p}, \boldsymbol{\omega}_o)}{dE(\boldsymbol{p}, \boldsymbol{\omega}_i)} = \frac{dL_o(\boldsymbol{p}, \boldsymbol{\omega}_o)}{L_i(\boldsymbol{p}, \boldsymbol{\omega}_i)(\boldsymbol{n}_p.\boldsymbol{\omega}_i)d\boldsymbol{\omega}_i} \tag{10}$$

The last step is obtained from (7).

The information in the BRDF can be used to derive the relationship between incident and reflected radiance. By manipulating and then integrating (10) over all directions, we can derive an expression for computing the reflected radiance at a point $\boldsymbol{p}$. We also note that the total radiance leaving a surface is the sum of the emitted radiance and the reflected radiance and also from (10) we get:

$$L_o(\boldsymbol{p}, \boldsymbol{\omega}_o) = L_e(\boldsymbol{p}, \boldsymbol{\omega}_o) + \int_{\Omega_i} L_i(\boldsymbol{p}, \boldsymbol{\omega}_i) f_r(\boldsymbol{\omega}_i, \boldsymbol{p}, \boldsymbol{\omega}_o) |\boldsymbol{\omega}_i.\boldsymbol{n}_p| d\boldsymbol{\omega}_i$$

Since each incidence radiance corresponds to an outgoing radiance, as expressed in equation (9), and also if $\boldsymbol{r}(\boldsymbol{p}, \boldsymbol{\omega})$ be the ray casting function then the gloabal illumination model of the famous rendering equation as described by James Kajiya in his paper 'The Rendering Equation'[6] in hemispheric form:

$$L_o(\boldsymbol{p}, \boldsymbol{\omega}_o) = L_e(\boldsymbol{p}, \boldsymbol{\omega}_o) + \int_{\Omega_i} L_o(\boldsymbol{r}(\boldsymbol{p}, -\boldsymbol{\omega_i}), \boldsymbol{\omega}_i) f_r(\boldsymbol{\omega}_i, \boldsymbol{p}, \boldsymbol{\omega}_o) |\boldsymbol{\omega}_i.\boldsymbol{n}_p| d\boldsymbol{\omega}_i \tag{11}$$

Kajiya suggested we follow a Monte Carlo approach to computing the equation (11) in a method that is now commonly called Path Tracing.

Although several top notch BRDFs exist (notably the Cook-Torrance model[1]), we will be using the Phong distribution:

$$f_r(\boldsymbol{\omega}_i, \boldsymbol{p}, \boldsymbol{\omega}_o) = k_d \frac{1}{\pi} + k_s \frac{n+2}{2\pi} \cos^n \alpha$$

where α = angle between the reflected ray and ideal specular reflection and n is the specular exponent. For preservation of energy, the restriction that $k_s + k_d \leq 1$ is essential.

5 Existing Systems

Several top notch ray tracing APIs already exist in the computer graphics world. Vulkan, the graphics + compute API has a ray tracing pipeline in one of its extensiosn. Advanced game engines like Unity handle ray tracing by themselves while other contemporary platform specific computer graphics APIs like DirectX have their own ray tracing pipeline. Obviously for physically based rendering the PBRT[8] is the best resource for beginners like ourselves to learn. A more convincing analysis of existing ray tracing APIs Ours is a simple ray tracer that implements the Whitted Ray Tracer Model but we are inclined to also have a little support for the Kajiya rendering equation.

6 System Block Diagram

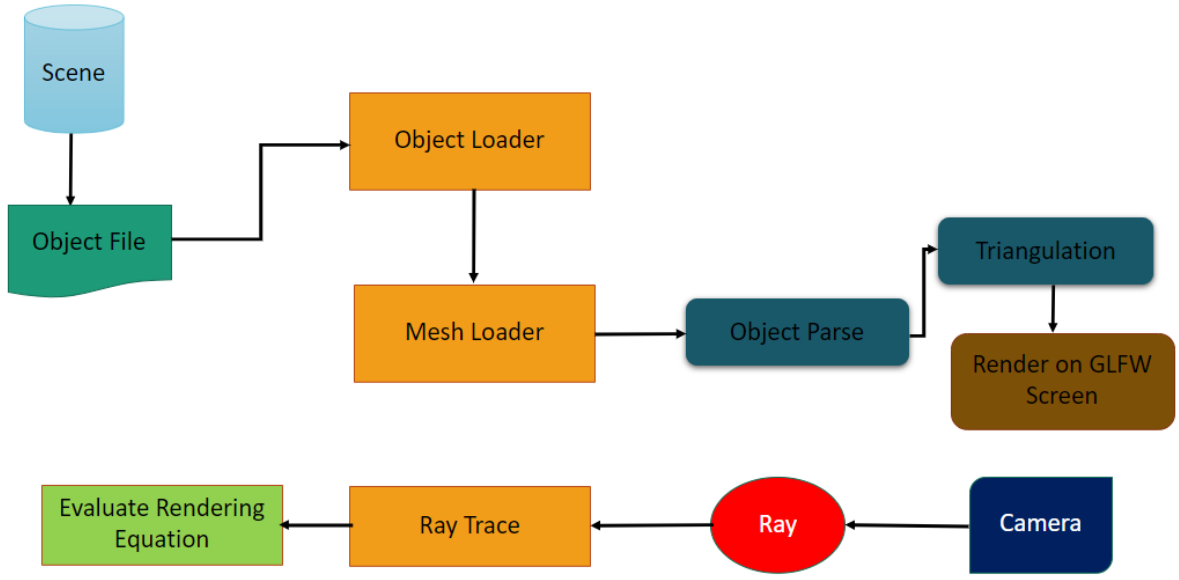

Figure 4: A simple block diagram representing the workings of Geocold

7 Project Schedule

The work on the project has already begun and we wish to finish the work as quickly as we can. The project is supposed to be quite a heavy one with tons of code for doing physically based rendering and also for actually rendering things on the screen using graphics APIs. The final project alongside with a report shall be submitted within the allocated time i.e., it should be finished by the end of July.

References

[1] R. L. Cook and K. E. Torrance. "A Reflectance Model for Computer Graphics". In: *ACM Trans. Graph.* 1.1 (Jan. 1982), pp. 7–24. ISSN: 0730-0301. DOI: 10.1145/357290.357293. URL: https://doi.org/10.1145/357290.357293.

[2] Gerald Farin. *Curves and surfaces for CAGD: A practical guide*. 5th ed. The Morgan Kaufmann Series in Computer Graphics. Morgan Kaufmann, 2001. ISBN: 0080503543.

[3] Ron Goldman. "Illicit expressions in vector algebra". In: *ACM Trans. Graph.* 4 (1985), pp. 223–243.

[4] Eric Haines and Tomas Akenine-Möller, eds. *Ray Tracing Gems*. http://raytracinggems.com. Apress, 2019.

[5] Wojciech Jarosz. "Efficient Monte Carlo Methods for Light Transport in Scattering Media". PhD thesis. UC San Diego, Sept. 2008.

[6] James T. Kajiya. "The Rendering Equation". In: *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '86. New York, NY, USA: Association for Computing Machinery, 1986, pp. 143–150. ISBN: 0897911962. DOI: 10.1145/15922.15902. URL: https://doi.org/10.1145/15922.15902.

[7] Eric P. Lafortune. "Mathematical Models and Monte Carlo Algorithms for Physically Based Rendering". In: 1995.

[8] Wenzel Jakob Matt Pharr Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. 3rd. Morgan Kaufmann Publishers, 2016. ISBN: 0128006455.

[9] Bui Tuong Phong. "Illumination for computer generated pictures". In: *Communications of the ACM* 18 (1975), pp. 311–317.

[10] Turner Whitted. "An Improved Illumination Model for Shaded Display". In: *Commun. ACM* 23.6 (June 1980), pp. 343–349. ISSN: 0001-0782. DOI: 10.1145/358876.358882. URL: https://doi.org/10.1145/358876.358882.