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 28, 2022

Accepted by: .....

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

Date of Submission: June 28, 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 28, 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.

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 Primitives

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([1]) and Goldman([2]).

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[1].

4.1 On Points and Vectors

Goldman (see [2]) 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[4] and the ones mentioned in Nvidia's Ray Tracing Gems[3]). We denote points with capital letters like P or Q, and we denote vectors with small boldface letters like $\boldsymbol{u}$ or $\boldsymbol{v}$.

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)

In C++, we can describe this behavior difference simply through the type system to create an API that would at the least follow basic axioms in mathematics and algebra.

To come up with a decent and mathematically correct ray tracer, it is essential that we use the C++ type system to handle basic principles of algebra, and analysis.

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 [4]) 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.

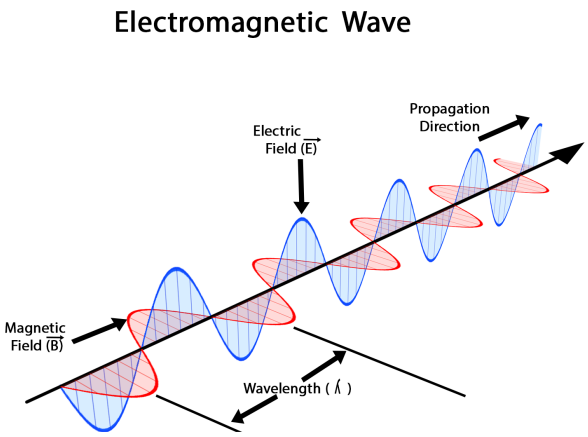

Figure 1: Light as an electromagnetic wave.

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. 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.

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.*

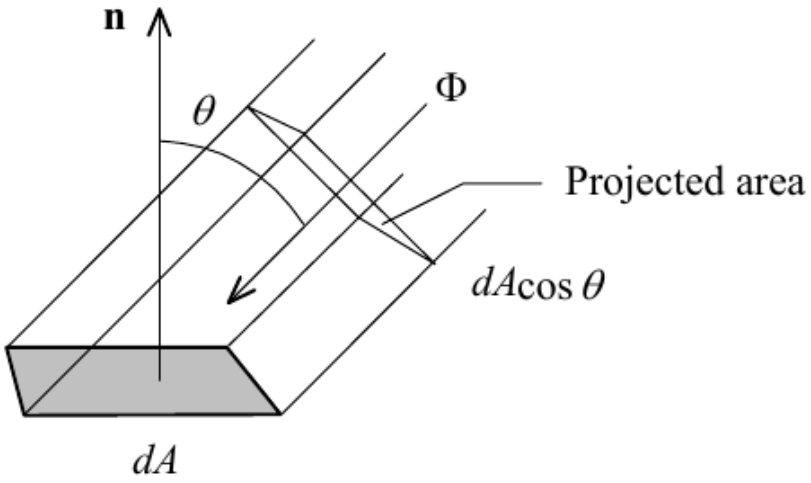

Figure 2: Ray of Light Intersecting a surface

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

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

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 $d\omega$.*

$$L = \frac{d^2\Phi}{dAcos\theta d\omega}$$

In order to avoid the complexity of having to take limits to deal with the radiance at above and below a surface, we define radiance arriving at a point on a surface and the radiance leaving that point.

Consider a point p on the surface of an object. Let $L_i(\boldsymbol{p}, \boldsymbol{\omega})$

References

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

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

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

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