Realistic Rendering with Photon Mapping for Indirect Illumination

Yi Ge
Yuan Liu
Rui Wang
Chenhan Lyu
Xiaoyu Liu

Instructor: Prof. James O'Brien

August 2020

Abstract

This thesis presents the concepts behind creating a successful image rendering engine using photon mapping. This thesis first talks about the introduction and method of our project. Then the theory used in this engine and the implementation of this engine will be discussed. Also, this thesis will display the performance of this engine. Finally, this thesis will show the result.

Contents

1 Introduction

Photon mapping is a two-pass global illumination algorithm developed by Henrik Jensen. It is an effective alternative to pure Monte Carlo ray tracing technology. Photon mapping decouples the illumination solution from the geometry and is represented in a spatial data structure called a photon map. This decoupling proved to be very powerful because the terms of the rendering equation can be calculated separately and stored in a separate photon map. This is also why photon mapping is very flexible because part of the rendering equation can be solved using other techniques. Photon mapping is also extended to take into account the participating media effects involved, such as sub-surface scattering and volume caustics.

The purpose of choosing this topic comes from our study of ray tracing. We realized that ray tracing has some drawbacks when dealing with indirect illuminations such as bleeding and caustics. Therefore, we would like to apply photon mapping to achieve more complicated rendering.

This thesis describes the process of implementation of the tools, the concept behind it, and the results achieved. Also, the assessment of how well our implementation works and future plan of our project will be included.

1.1 Objectives

The following objectives list was set at the beginning of the project and its success will be analysed at the end of the thesis.

• **Complete integrative tool** Our objective is to build an image rendering engine that integrates ray casting, ray tracing, photon mapping, and progressive photon mapping. The engine can render 3D images based on the data describing the scene and export them in a ppm file

• **Make up for the shortcomings of ray tracing** Since ray-tracer occurs to have some disadvantages when dealing with indirect illuminations such as bleeding and caustics, we would like to apply photon mapping to achieve more complicated rendering.

1.2 Thesis Overview

This thesis is divided in the following way:

• **Methods:** Briefly describes the basic idea of our project

• **Theory:** Presents the knowledge background and details on the theory behind this implementation. This section describes related formula of photon mapping and progressive photon mapping, the theory of Monte Carlo algorithm, and the principle of kd-tree

• **Implementation:** Presents the tools and libraries that used in this project. In addition, this section provides the class dependency diagram of this project and the algorithms used to achieve it. Includes as well as the pseudo code and the application of kd-tree

• **Evaluation and Testing:** This section tests whether kd-tree works properly in different test environments. In addition, the performance of the program with and without multi-thread will be tested
• **Experiment Results and discussion:** This section uses different algorithms to render our models and visualizes the photon map to help see what role it plays in shading. Also, this thesis tests the result with different numbers of total photons in photon mapping and makes a contrast between basic photon mapping and progressive photon mapping.
• **Future work:** Presents the theory outlook of importance sampling and other theories that can improve our project.
• **Labor Division:** Describe the contribution of each team member

2 Method

2.1 First Pass-Photon Tracing

•**Photon Emission:** A photon's life begins at the light source. For each light source in the scene, we create a set of photons and divide the overall power of the light source amongst them. Brighter lights emit more photons then dimmer lights. Finding the number of photons to create at each light depends largely on whether decent radiance estimates can be made during the rendering pass. For good radiance estimates the local density of the photons at surfaces needs to provide a good statistic of the illumination.
• **Photon Scattering:** Emitted photons from light sources are scattered through a scene and are eventually absorbed or lost. When a photon hits a surface we can decide how much of its energy is absorbed, reflected and refracted based on the surface's material properties
• **Photon Storing:** For a given scene we may shoot millions of photons from the light sources. It is desirable that our photon map is compact to reduce storage costs. We also want it to support fast three-dimensional spatial searches as we will need to query the photon map millions of times during the rendering phase.

2.2 Second Pass-Rendering

The rendering engine is a distribution ray tracer in which rays are traced from the eye into the scene. As the rays are traced through several reflections, their contribution to the final pixel radiance becomes lower. We then apply the approximate estimate which for all surfaces equals a radiance estimate obtained from the photon map. This computation is performed using importance sampling where the information about the incoming flux is integrated with the BRDF to provide optimized sampling directions.

3 Theory

3.1 Photon Mapping

Recall the two-pass algorithm, the first is photon tracing pass, where photons would be traced from the light sources into the scene and stores them in a photon map as they interact with the surfaces. The second pass is rendering in which the photon map is used to estimate the illumination in the scene. Given a photon map, exitant radiance at any surface location p can be estimated as

$$R(p, \vec{d}) \approx \sum_{i=1}^{n} \frac{F_r(p, \vec{d}, \vec{d_i})\phi_i(p_i, \vec{d_i})}{\pi r^2} \tag{1}$$

where n is the number of nearest photons used to estimate the incoming radiance, ϕ_i is the flux of the ith photon, F_r is the bidirectional reflectance distribution function (BRDF), $\vec{d}$ and $\vec{d_i}$ are the outgoing and incoming directions, r is the radius of the sphere containing the n nearest photons.

As the photon density increases, the radiance estimate will converge to the correct solution, which makes photon mapping a consistent algorithm. To ensure convergence to the correct solution, in our implementation, we would use large number (e.g. 1000,000) of photons in the photon map and radiance estimation.

3.1.1 Ray Tracing Process

During ray tracing pass, we aim to find all the surfaces in the scene visible through each pixel in the image. Each ray path includes all specular bounces until the first non-specular surface seen. For each ray path, we sotre all hit points along the path where the surface has a non-specular component in the BRDF. For each hit point, we store the hit location p, the ray direction $\vec{d}$, scaling factors including BRDF and pixel filtering value, and the associated pixel location. We represent these values in the following structure:

hitpoint{

position p	*Hit location*
vector $\vec{n}$	*Normal at p*
vector $\vec{d}$	*Ray direction*
double $BRDF$	*BRDF index*
double x, y	*Pixel location*
color rgb	*Pixel weight*

}

3.1.2 Photon Tracing Process

The photon power would be accumulated at the hit points found in tay tracing. It can be divided into multiple passes where each pass consists of

tracing a given number of photons into the scene in order to build a photon map. After each photon tracing pass, we loop through all hit points and find the photons within the radius of each hit point. We use the newly added photons to refine the estimate of the illumination within the hit point as described in the following section. Once the contribution of the photons have been recorded they are no longer needed, and we can discard all photons and proceed with a new photon tracing pass. This continues until enough photons have been accumulated and the final image quality is sufficient. Then we can render an image after each photon tracing pass.

3.2 Progressive Photon Mapping

The main idea in progressive photon mapping is to reorganize the standard photon mapping algorithm based on the conditions of consistency, in order to compute a global illumination solution with arbitrary accuracy without storing the full photon map in memory. Progressive photon mapping is a multi-pass algorithm in which the first pass is ray tracing and all subsequent passes use photon tracing. Each photon tracing pass improves the accuracy of the global illumination solution and the algorithm is progressive in nature.

3.2.1 Photon Tracing Process

In addition, we store extra data necessary for the progressive radiance estimate including a radius, the intercepted flux, and the number of photons within the radius. We represent these values in the following structure:

hitpoint{

position p	*Hit location*
vector $\vec{n}$	*Normal at p*
vector $\vec{d}$	*Ray direction*
double $BRDF$	*BRDF index*
double x, y	*Pixel location*
color rgb	*Pixel weight*
double r	*Current photon radius*
integer n	*Accumulated photon count*
double τ	*Accumulated reflected flux*

}

3.2.2 Photon Tracing Process

As more photons are accumulated, the quality of the image will progressively improve toward the final result.

3.2.3 Radiance Calculation

After each photon tracing pass we can evaluate the radiance at the hit points. Recall that the quantities stored include the current radius and the current intercepted flux multiplied by the BRDF. The PPM radiance is multiplied by the pixel weight and added to the pixel associated with the hit point. The radiance is evaluated as follows:

$$\begin{aligned} R(p,\vec{d}) &= \int_{2\pi} F_r(p,\vec{d_{out}},\vec{d_{in}})R(p,\vec{d_{in}})(\vec{n}\vec{d_{in}})dd_{in} \\ &\approx \frac{1}{\delta Area}\sum_{i=1}^{n} F_r(p,\vec{d},\vec{d_i})\phi_i(p_i,\vec{d_i}) \\ &= \frac{1}{\pi Radius(p)^2}\frac{\tau(p,\vec{d})}{N} \end{aligned} \tag{2}$$

Where d_{in} and d_{out} correspond to incoming and outgoting direction, $\tau(p,\vec{d})$ represents the flux with BRDF sotred, N is the number of total emitted photons in order to normalize $\tau(p,\vec{d})$. During progressive photon mapping, each hit points has a radius $Radius(p)^2$, and the algorithm is to reduce the raduis while increasing the number of photons accumulated within this radius. The progressive radiance estimation ensures that the photon density at each hit point increases at each iteration and ensure the consistency of Equation (1).

3.3 Texure Mapping

In addition to our implementation of photon mapping (PM) and progressive photon mapping (PPM), other effects present in our scenes is texture mapping. Texture mapping was basically fully functional, but we did make good use of it by adding effects on the inside of the sphere and other shapes with backward, lightened images to simulate being able to see the reflection of marble and floor texture. The radiance of the surface is stored in reconstruction texture maps applied to the surfaces. The texture maps are approximated illumination maps (AIMs).

4 Implementation

4.1 Utility Data Structure

Vector3: A tool designed to store the 3D position, direction, supporting addition, subtraction, multiplication, and division operations, where A * B represents A cross multiply B.

Color: A tool designed to store the RGB color: the brightness of the three primary colors (red, green, blue), value range 0 to 1, supporting some fundamental operations.

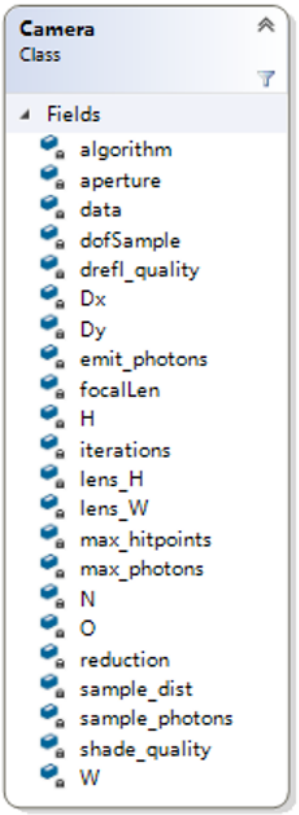
ppm: A tool dealing with input and output which can be used to read the ppm file as the texture and output the result to a ppm file.

ObjReader: A tool handling the input of the obj file.

4.2 Key Data Structure

4.2.1 Camera

Camera class is the viewpoint, which is very important in direct illumination. Several visual parameters are set here including some const numbers.



***const string STD_Algorithm*:** rendering algorithms used in standard cases

***const double STD_Aperture*:** the aperture size under standard conditions, the depth of field effect is rendered when the aperture size is greater than 0

***const int STD_IMAGE_WIDTH(Height)*:** pixel length (or width) of a standard photo

const int STD_SHADE_Quality*:16=Under standard conditions, Monte Carlo calculation times when calculating shadows

***const int STD_DREFL_Quality*:** Under standard conditions, Monte Carlo calculation times when calculating specular diffuse reflection

***const int STD_MAX_Hitpoints*:** capacity of collision point map in standard case

***const int STD_Iterations*:** iterations of PPM algorithm in standard case

***const int STD_Reduction*:** the convergence coefficient of the convergence radius of PPM algorithm in the standard case

***const int STD_MAX_Photon*:** graph capacity in standard case

***const int STD_EMIT_Photons*:** number of photons emitted under standard conditions

***const int STD_SAMPLE_Photons*:** number of sampled photons in standard case

***const double STD_SAMPLE_Dist*:** sampling photon radius in standard case

4.2.2 Material

The material class describes some properties of a certain surface.

***color, absor*:** The color of objects and the shade absorbed by transparent objects

***refl, refr, diff, spec*:** percentage of reflection, refraction, diffuse, and specular diffuse

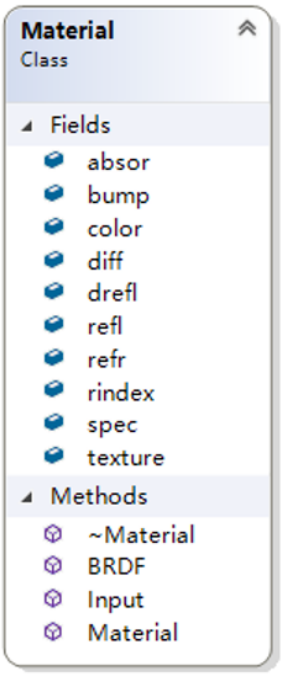
***rindex*:** refractive index

***drefl*:** the deviation length of the reflected light when using Monte Carlo algorithm simulates specular diffuse reflection (does not use this algorithm when the deviation length = 0)

***texture*:** texture related PPM files (= null without texture)

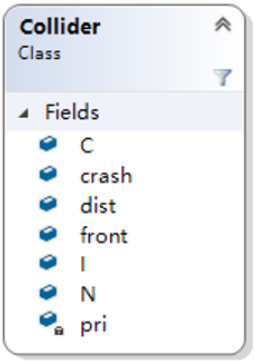
***input (string VaR, stringstream)*:** reads data from the stream, where var prompts the variable to be read

***BRDF(Vector3 ray_R, Vector3 N, Vector3 ray_I)*:** Diffuse lighting model. ray_R is the direction of the light source. N is the normal vector at the collision point. ray_I is the direction of the detection ray, it is effective when the angle between ray_R, ray_I, and N is less than 180 degrees



4.2.3 Collider

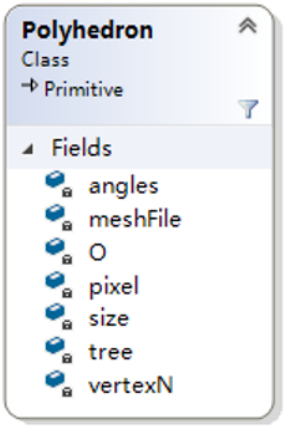
The Collider class records the primitive that ray hits



***pri*:** collided object
***dist*:** the distance from the ray to the collision point
***crash*:** whether it hits
***front*:** collides with the surface in the direction of the normal vector of the object
***N*:** The normal vector of the object at the collision point
***C*:** collision point
***I*:** the direction of the incident ray

4.2.4 Primitive

The Primitive class who has five subclasses: Plane, Polyandron, Rectangle, Sphere, Triangle. They are used to describe the objects in the scene. When a scene is created, all these kinds of primitives are stored in a linkedlist.



Among them, **Polyandron** is the object read from an obj file. Inside it, there is a tree structure to store all the triangle patches of the object

Polyandron
***O*:** coordinates of 3D model center

***size*:** the ratio of scaling the original obj model on three axes

***angles*:** The angle at which the original obj model is rotated on the three coordinate axes

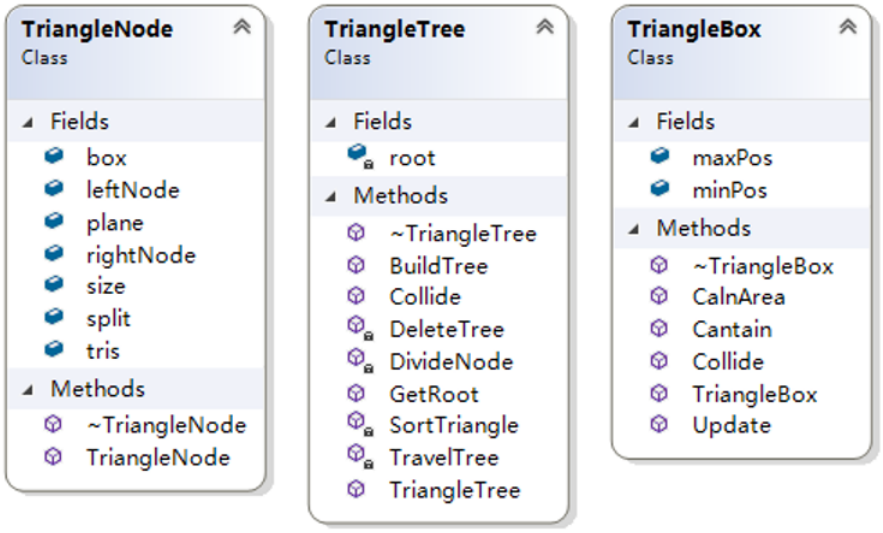
***vertexnN*:** all normal vectors in 3D model

***pair*{ *double, double* } **pixel*:** all texture points in 3D model

***tree*:** Kd-tree for organizing triangular faces. It is built to accelerate the intersection routine

4.2.5 triangleTree

We wrap each triangle patch in a **TriangleBox**, and then a balanced Kd-tree is built to accelerate the intersection routine. **TriangleNode** is a data structure used to describe a node of the **TriangleTree**.



Class trianglebox (bounding box class of triangle kd-tree node)
***minPos*:** minimum value of bounding box on three axes

***maxPos*:** the maximum value of the bounding box on the three axes

***Update*:** (Triangle* tri): maintain the bounding box for a newly added triangle

***Cantain (Vector3 O)*:** judge whether a point is in the bounding box

***CalnArea ()*:** the surface area of the bounding box

***Collide (Vector3 ray_O, Vector3 ray_V)*:** The ray (ray_O, ray_V) intersects the bounding box and returns the distance, they don't intersect when the distance is less than 0

Class trianglenode (node class of triangle kd-tree)
***tris*:** all triangles on the node
***size*:** the number of triangles on the node
***plane*:** divided plane
***split*:** coordinate value of division
***box*:** bounding box of node
***leftNode*:** left child node
***rightNode*:** right child node

Class triangletree (triangle KD tree class)
***root*:** root node

***DeleteTree (TriangleNode* node)*:** recursively delete the class tree, and currently recurse to the node node

***SortTriangle(Triangle** tris,int l,int r,int coord,bool minCoord)*:** Sort the triangles of tris[l r], coord is the key axis, minCoord indicates whether to sort the nodes with the smaller coordinate value of the axis

***DivideNode (TriangleNode* node)*:** divide nodes

***TravelTree (TriangleNode* node, Vector3 ray_O, Vector3 ray_V)*:** Ray (ray_O, ray_V) traverses kd-tree and reaches node node

4.2.6 Light

The Light class who has two subclasses: AreaLight and PointLight. When a scene is created, all the lights will be stored in a linkedlist called Light.

***sample*:** is a random number for each light source which is used in the hash to judge whether there may be aliasing during rendering.

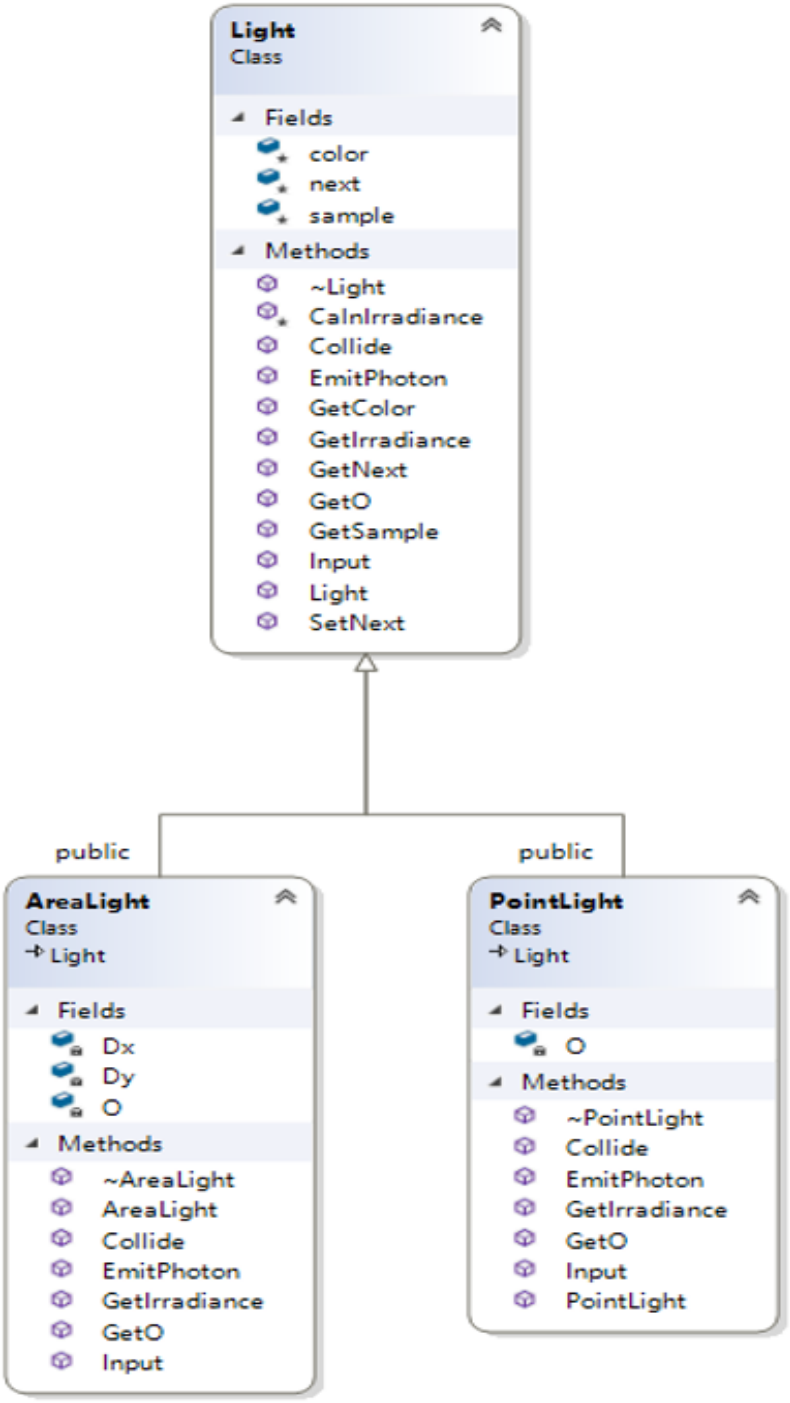
***crash_dist*:** is the distance traveled by the light before collision.

***Collide(Vector3 ray_O, Vector3 ray_V)*:** calculate the collision of light (ray_O, ray_V) to the item, the distance traveled before the collision is stored in crash_dist, and return whether there is a collision.

***Color GetIrradiance (Primitive* pri, Primitive* primitive_head, int shade_quality, int* hash)*:** calculate the ratio of the illuminance of the collision point multiplied by the diff or spec, shadow needs to be considered.

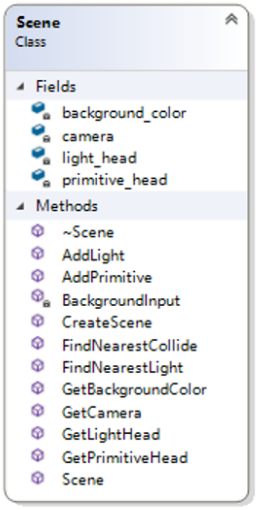
***Photon EmitPhoton ()*:** randomly get a photon that has just been emitted.

***Vector3 Dx, Dy*:** the coordinates of the light source, its length is equal to the length of half-axis length


Light
Class
Fields
color
next
sample
Methods
~Light
CalnIrradiance
Collide
EmitPhoton
GetColor
GetIrradiance
GetNext
GetO
GetSample
Input
Light
SetNext
public
public
AreaLight
Class
Light
Fields
Dx
Dy
O
Methods
~AreaLight
AreaLight
Collide
EmitPhoton
GetIrradiance
GetO
Input
PointLight
Class
Light
Fields
O
Methods
~PointLight
Collide
EmitPhoton
GetIrradiance
GetO
Input
PointLight

4.2.7 Scene

The Scene class describes the surroundings that we want to render. Once a scene is initiated, two linkedlists are created, one is used to store lights, and another is used to store primitives. What's more, there should be one and only one camera in a scene. ***primitive_Head*:** the chain header of an item set



***light_Head*:** the chain head of the light source set

***camera*:** pointer of camera

***background_Color*:** background color

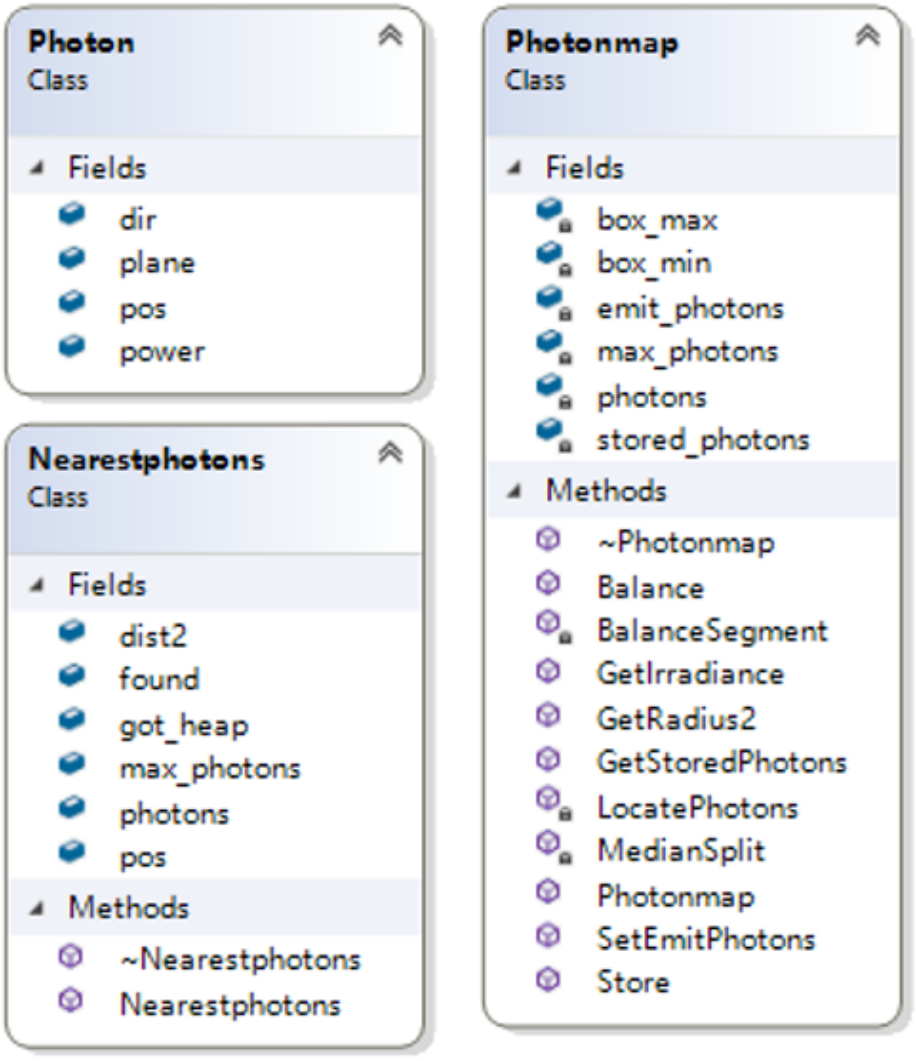
***create scene(string file)*:** read the scene data from the file

***FindNearestPrimitive (Vector3 ray_ O, Vector3 ray_ V)*:** Calculate the first object hit by the ray (ray_O, ray_V) and return it in the Collider package

***FindNearestPrimitive(Vector3 ray_O, Vector3 ray_V)*:** Calculate the first light source hit by the ray (ray_O, ray_V) and return it in the Collider package

4.2.8 Photon, Photonmap, Nearestphotons

Photon, Photonmap, ans Nearestphotons are three data structures used in photon mapping. There is a list which is empty at first. After the **Photons** are being emitted from the **Lights**, they are recorded in the list each time they hit a **Primitive**. After that, we get a photon list. To accelerate the sorting, we build a kd-ree named **Photonmap**. A **Nearstphotons** is a max heap used to find the k Photons closed to the **collider**.



Class Photonmap

***pos, dir*:** photon collision position and incident direction

***power*:** photon energy

***plane*:** the plane by which this photon is divided in the kd-tree

***BalanceSegment (Photon* porg, int index, int st, int en)*:** balance the photons of porg[st en]

***MedianSplit(Photon* porg,int st,int en,int med,int axis)*:** divide the photon of porg[st en] with axis as the axis and med as the center

***LocatePhotons(Nearestphotons* np,int index)*:** find the nearest k photons, the input and output data are all in np

Algorithm 1 Pseudocode for balancing photonmap

```
function KDTREE *MEDIANSPLIT(points)
    Find the bounding box of all the photons.
    Use the plane of the longest axis
    Find the median of the points in this plane
    s1 = all points below median
    s2 = all points above median
    node = median
    node.left = MedianSplit (s1)
    node. right = MedianSplit(s2)
    return node
```

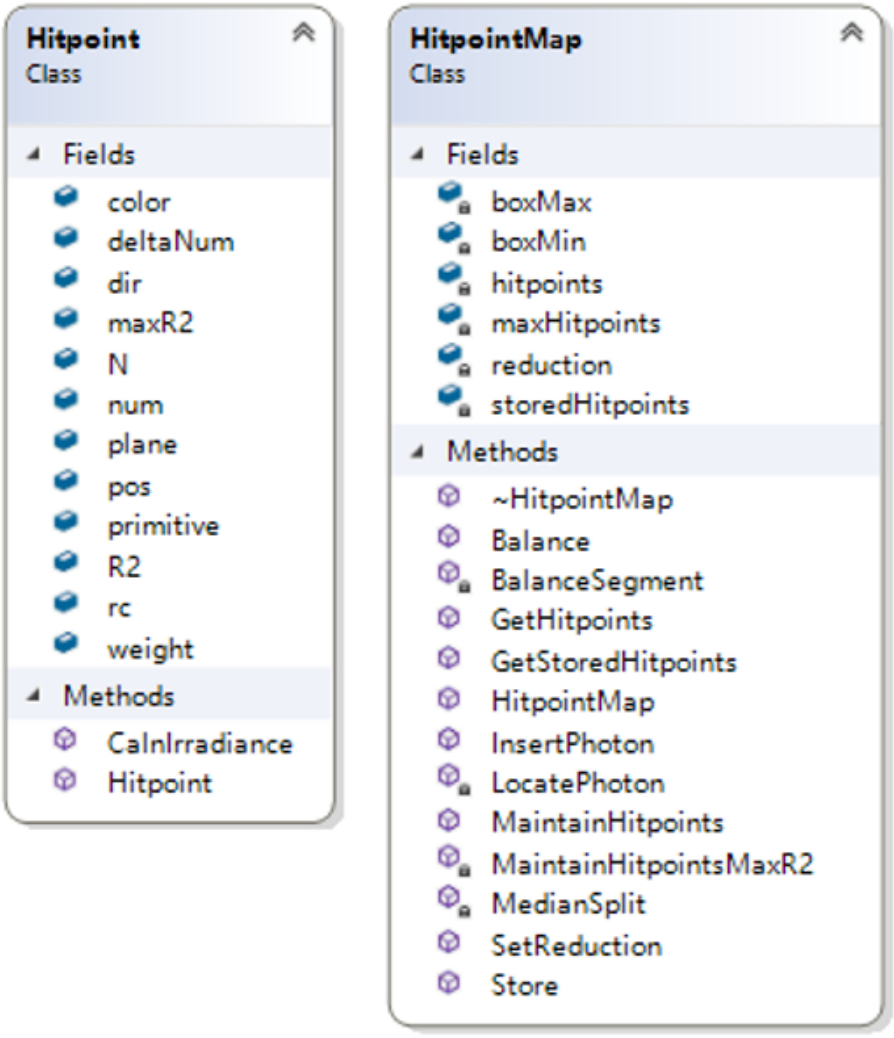
A max heap is an efficient way of keeping track of the element that is furthest away from the point. When the max heap is full, we can use the distance d to the root element to adjust the range of the query. Thus, we skip parts of the kd-tree that are further away than d.

Algorithm 2 Pseudocode for locating the nearest photons in the photon map

```
function LOCATE PHOTONS(p)
    np = number of photons
    if 2p + 1 < np then
        Compute distance to plane
        delta = signed distance to splitting plane of node n
        if delta < 0 then
            locate photons (2p)
            if delta^2 < d^2 then
                locate photons (2p + 1)
            else
                locate photons (2p + 1)
                if delta^2 < d^2 then
                    locate photons(2p)
                    delta2 = squared distance from photon p to x
                    if delta2 < d^2 then
                        insert photon into max heap h
                        distance = squared distance
                              to photon in root node of h
                        d^2 ← distance
```

4.2.9 Hitpoint, HitpointMap:

There are two new data structures used in PPM. PPM first emits ray from the camera, and each pixel stores all the **Hitpoints** at which ray from that pixel intersects the non-shiny surface. **HitpointMap** is designed to store and maintain all the **Hitpoints**.


Hitpoint
Class
Fields
color
deltaNum
dir
maxR2
N
num
plane
pos
primitive
R2
rc
weight
Methods
CalnIrradiance
Hitpoint
HitpointMap
Class
Fields
boxMax
boxMin
hitpoints
maxHitpoints
reduction
storedHitpoints
Methods
~HitpointMap
Balance
BalanceSegment
GetHitpoints
GetStoredHitpoints
HitpointMap
InsertPhoton
LocatePhoton
MaintainHitpoints
MaintainHitpointsMaxR2
MedianSplit
SetReduction
Store

Class Hitpoint

***pos, dir*:** the position and incident direction of the collision point

***N*:** the normal vector of the collision point

***primitive*:** the object where the collision point is located

***rc*:** image pixels that rc corresponded to (rc/W, rc%W)

***weight*:** the weight of the collision point to the pixel

***plane*:** the plane divided by the collision point in the kd-tree

***R2, maxR2*:** the radius of the sphere containing num photons at the collision point, and the maximum value of the radius of the node under the subtree

***num, deltaNum*:** The collision point contains num photons and the number of photons added in the new round of photon mapping

***color*:** the shade calculated at the collision point

***CalnIrradiance (Photon*)*:** calculate the influence of a photon on the shade of the collision point

Class HitpointMap

***maxHitpoints, storedHit*:** max collision points and stored collision points

***hitpoints*:** all storage collision points

***boxMin, boxMax*:** two angles of the spatial range

***reduction*:** coefficient in double reduction; ppm algorithm

***BalanceSegment (Hitpoint* horg, int index, int st, int en)*:** The collision point of equilibrium horg [st $\sim$ en]

***MedianSplit (Hitpoint* horg, int st, int en, int med, int axis)*:** Taking axis as the axis and Med as the center, the collision points of horg [st $\sim$ en] are divided

***maintainhitpointsmaxr2()*:** maintains the maxr2 of each node in the tree

***locatephoton (photon * photon, int index)*:** count the photon into the collision point containing it

***store (hitpoint)*:** stores a collision point in the collision point graph

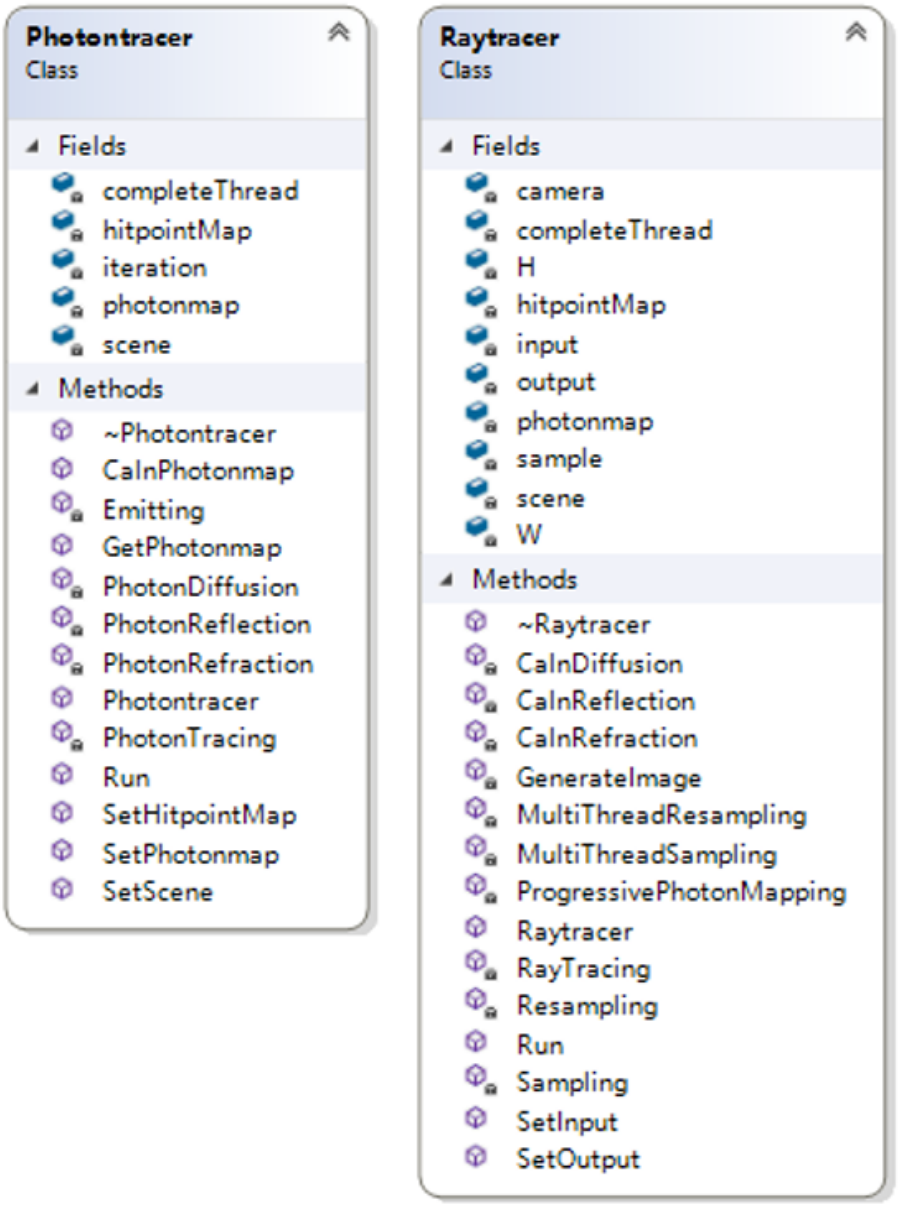
***maintainhitpoints ()*:** these collision points are maintained and called once every iteration in PPM algorithm to complete radius attenuation

***balance ()*:** build a balanced KD tree, which is used after the collision point is stored and before query

***InsertPhoton (Photon)*:** A photon is inserted into the collision point graph to calculate the influence of the photon on the image

4.3 Drivers

Photontracer, Raytracer: two drivers using the data structures listed in 4.1 and 4.2 to render the entire scene.



4.3.1 Class RayTracer

***CalnDiffusion (Collider* collider, int* hash, int rc, Color weight)*:** Calculate the diffuse color light, hash is used to record the items that the light is passing through (the adjacent pixels with different hash values should be deserrated using super sampling), rc is the corresponding pixel point, weight is the weight of this ray to the pixel

***CalnReflection (Collider* collider, Vector3 ray_V, int dep, bool refracted, int* hash, int rc, Color weight)*:** calculates the reflected color light, in which ray_V is the direction of the incident ray and DEP is the number of iteration layers. Refracted represents whether the ray has been refracted odd times

***CalnRefraction (Collider* collider, Vector3 ray_V,int dep,bool refracted,int* hash,int rc,Color weight)*:** Calculate the color light obtained by refraction

***RayTracing (Vector3 ray_O,Vector3 ray_V,int dep,bool refracted,int* hash,int rc,Color weight)*:** The main iterative function of ray tracing connects different levels of rays

***Sampling(int threadID,int randID)*:** Raytracing sampling, threadID thread number, randID as random number seed

***Resampling(int threadID,int randID)*:** Raytracing Super Sampling (desaliasing)

***MultiThreadSampling(int randIDBase = 0)*:** Tissue multithreaded ray tracing sampling (random seeds taken from randIDBase)

***MultiThreadResamplling(int randIDBase = 0)*:** Tissue multithreaded ray tracing super sampling (random seeds taken from randIDBase)

***ProgressivePhotonMapping(int SPPMIter = 0)*:** Iterative process in PPM algorithm

***GenerateImage(std::string file)*:** Export the camera data to the File image file

4.3.2 Class PhotonTracer

***PhotonTracing(Photon,int dep)*:** The main iterative function of photon mapping plays a cohesive role

***bool PhotonDiffusion(Collider* collider, Photon, int dep, bool refracted, double* prob)*:** It's a simulation to determine whether photons are refracted (probability event).

***PhotonReflection(Collider* collider,Photon,int dep,bool refracted, double* prob)*:** Determine if the photon is reflecting

***PhotonRefraction(Collider* collider,Photon,int dep,bool refracted, double* prob)*:** Determine if the photon is refracted

***Emitting(int threadID,int randID)*:** performs multithreaded photon mappings, threadID is the thread number, and randIDis the random number seed

***CalnPhotonmap()*:** The photonic map is calculated based on the scene.

***Run(int randIDBase = 0)*:** Start the operation and calculate the photon map based on the scene (random seeds from randIDBase)

5 Evaluation and Testing

5.1 Kd-tree

Test the kd-tree class if it works properly.

5.1.1 Tested Products

The "TriangleBox" class in the TriangleTree.cpp file.
The "TriangleNode" class in the TriangleTree.cpp file.
The "TriangleTree" class in the TriangleTree.cpp file.

5.1.2 Test Environment

Table 1: Test Environment

OS	Compiler	IDE	CPU	RAM
Windows 10 1903	Mingw-w64 7.0	Visual Studio	Intel Core i7-9700K	16GB
macOS Catalina 10.15.6	AppleClang 11.0.3	Visual Studio Code	Intel Core i5-1038NG7	16GB
Ubuntu 16.04	GNU 10.2	Visual Studio Code	Intel Core i7-9700K	16GB

5.1.3 Test Method

Unit-test
Integration test

5.1.4 Test Process

Test TriangleBox TriangleBox is a simple class as it only contains the maximum and minimum three-dimension vector. We perform several tests against the class and its stability, which are written below.

•TEST(TriangleBoxSanityTest, CanConstrctorInDefault)
Construct the TriangleBox;
Check if the maximum value matches the default value;
Check if the minimum value matches the default value;

This test checks if the TriangleBox can be created correctly.

•TEST(TriangleBoxSanityTest, CanUpdate)
Construct the TriangleBox;
Construct the Triangle;
Using the Triangle to Update the Value;

Check each value matches the expectation;

This test checks if the TriangleBox could successfully update its value after calling the function, we also test it with different Triangle, and also get the correct answer.

•TEST(TriangleBoxSanityTest, ContainWorks)

Construct the TriangleBox;
Construct the Triangle;
Using the Triangle to Update the Value;
Construct the 3D-vector;
Call the contain member function;
Check the result matches the expectation;

•TEST(TriangleBoxSanityTest, AreaAndCollideWorksExcept)

Construct the TriangleBox;
Construct the Triangle;
Using the Triangle to Update the Value;
Construct two 3D-vector for collide;
Check if the Area matches the expectation;
Check if the double return from collide matches the expectation;

The above test cases are run without problem.

Test TriangleTree

The TriangleTree class the main kd-tree class. The test process is complicated as the class is complicated. Fortunately, we were able to test it step by step and construct the class step by step.

•Can Construct and Delete

Construct the TriangleTree inside of a function, add some elements and return the function. These steps caused no error and the money have no leaks which checks the TriangleTree can be created correctly.

•Kd-Tree Works and Improve the efficiency

We included the ctime and recorded the start time run the code with iteration and then compared the result with the time using our tree enabled instead. The following table 1 is the test result with different number of elements. The following chart is the average time recorded with unite second.

•TEST(TriangleTreeSanityTest,TravelTest)

Construct the TriangleTree;
Build the Tree;
Random choose a node;
Call the travel function, get the Collier;
Verify the Collier information;

Table 2: Efficiency table

Element number	Iteration	Tree
10,000	23	12
20,000	35	17
40,000	46	28

5.1.5 Conclusion for the tests for kd-Tree

All tests passed for a certain rate of success, thus, the team believe the kd-tree works as wanted.

5.2 Test Multi-thread

5.2.1 Tested Products

The whole program

5.2.2 Test Environment

Table 3: Test Environment

Operating System	Compiler	IDE	CPU	RAM
Windows 10 1903	Mingw-w64 7.0	Visual Studio	Intel Core i7-9700K	16GB
macOS Catalina 10.15.6	AppleClang 11.0.3	Visual Studio Code	Intel Core i5-1038NG7	16GB
Ubuntu 16.04	GNU 10.2	Visual Studio Code	Intel Core i7-9700K	16GB

5.2.3 Test Method

Performance Testing
Integration test

Table 4: Test Results (Windows Platform)

Mode	Average time with Threads	Average time without Thread
RT	449.229s	1324.48s
PM	457.427s	1387.94s
PPM Iter=2000	1004.26s	2872.67s

5.2.4 Results

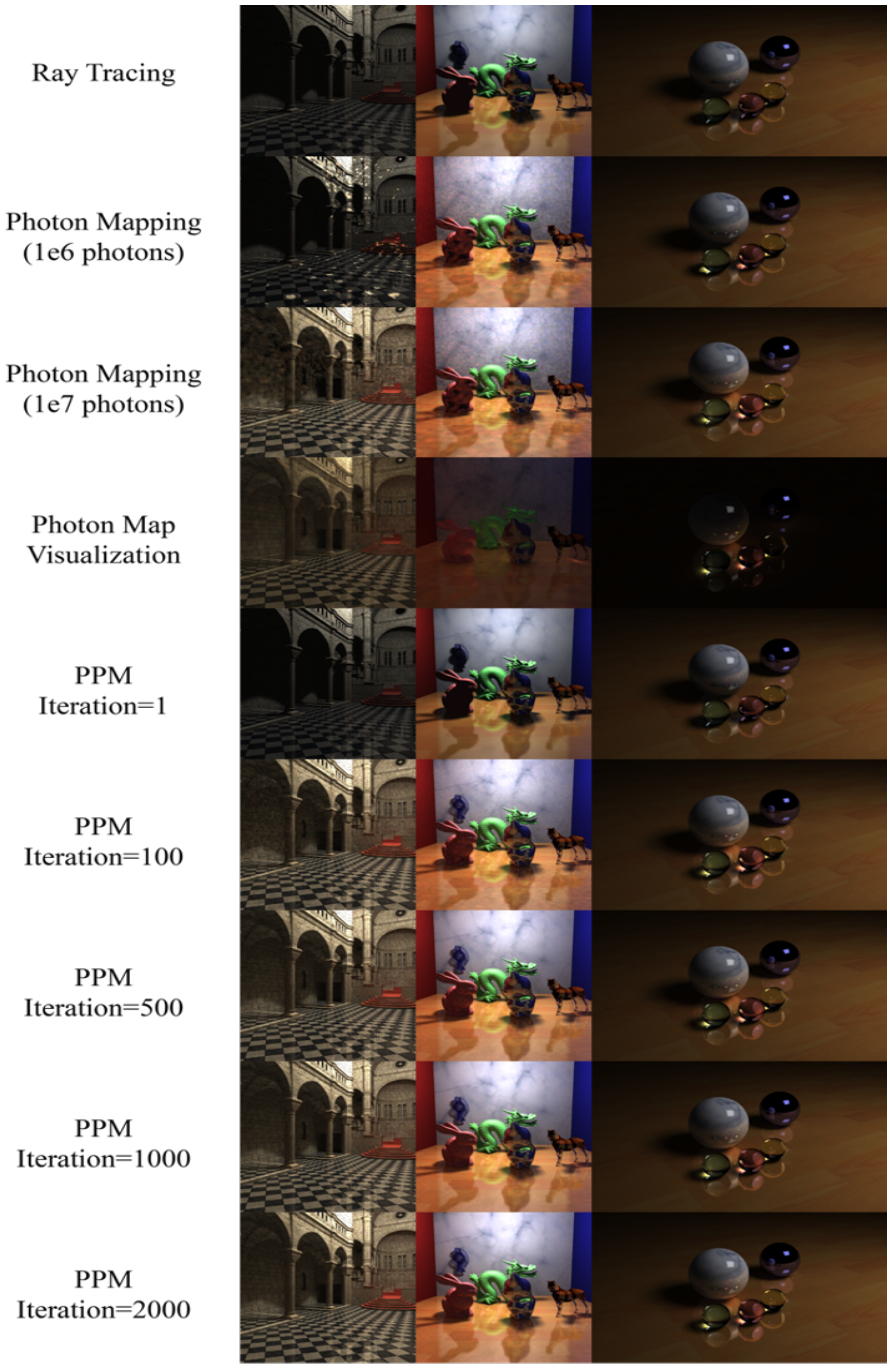
As shown in the above chart, the multi-tread dramatically increase the speed of rendering. However, the above result is only for the windows platform as we fail to finish the multi-thread in UNIX-based systems, like macOS and Ubuntu. In those platforms, the multi-thread algorithm is not working properly. However, according to Gerard J. Holzmann, all software will have bugs, as the code gets longer, the probability of having bugs is larger. (Holzmann, 2015) We cannot eliminate the bugs completely, but we will made our effort to solve them in the future work.

6 Experiment Results and Discussion

Table 5: Time Result for Algorithms

Total Emitted Photons	Max Sample Photons	Algorithm	Model 1	Model 2	Model 3
1e6	10	RT	676.572s	449.229s	19.457s
1e6	10	PM	670.657s	457.427s	20.495s
1e7	500	PM	722.575s	489.547s	20.902s
1e6	10	PPM Iter=100	658.041s	537.225s	44.847s
1e6	10	PPM Iter=500	1262.255s	751.345s	119.722s
1e6	10	PPM Iter=1000	1829.251s	998.553s	214.063s
1e6	10	PPM Iter=2000	3034.224s	1004.26s	413.245s

We used different algorithms to render our models including ray tracing, photon mapping and progressive photon mapping for different iterations and the time they consume are above. Also, we visualized the photon map to help see what role it plays in shading. The figure shows our results. Compared with photon mapping, ray tracing shows two drawbacks. Firstly, As is can be seen in model 2, surfaces of the red rabbit which cannot be shot hit by the lights are completely dark. However, in reality we should still see it because of the indirect illumination. Secondly, ray tracing cannot render caustics while photon mapping performs well on it. Our visualization of photon map illustrates that photon mapping solve the two shortcomings mentioned above. In PM, we also change the number of total photons and the number of sampled photons at each hit point. While doing so, we get different outcomes. With only 100,000 photons in total and 10 ones for each sampling, images are darker than those of 1,000,000 photons and 500 ones sampled. . And because of the insufficiency of photons, they show a lot of mottles. In addition, we made a contrast between basic photon mapping and progressive photon mapping. For only one iteration, PPM performs nearly the same as ray tracing. This happens because a very limited number of photons were emitted. Thus details were not well shown. With more photons in the scene, more indirect light rays will be taken into account and more details were displayed. PPM takes more time than PM but PPM sharpens the edges. In model 1 for PPM, the edges between two bricks are much clearer than those for PM. We also tried to use SPPM but didn't work it out successfully.


Ray Tracing
Photon Mapping (1e6 photons)
Photon Mapping (1e7 photons)
Photon Map Visualization
PPM Iteration=1
PPM Iteration=100
PPM Iteration=500
PPM Iteration=1000
PPM Iteration=2000
Model 1
Model 2
Model 3

7 Future Work

Until now, we finished ray casting, ray tracing, photon mapping and progressive photon mapping pipeline. Later, we will research on stochastic photon mapping and importance sampling method for sampling of the photons. The importance sampling could improve the Monte Carlo ray tracing in which a rough estimate of the irradiance based on the photon map is combined with the local reflection model to construct more efficient probability density functions. The sampling could dependent on the cached approximation of the incoming radiance, or scattering direction from a wide range of other BRDFs as well.

We will also develop a robust multithreading codebase for algorithm optimization. We should ensure that our method could word under different operating system including Linux, Unix and MacOS.

8 Labor Division

8.1 Code

Class Completion	
Yi Ge (Ellen)	Primitive, Light, Photonmap, Photontracer, Raytracer, TriangleTree, Hitpointmap
Yuan Liu (Hive)	Camera, Photonmap, Photontracer, Raytracer, TriangleTree, Hitpointmap
Rui Wang (Raymond)	Color, Ppm, Primitive, Light, Photonmap, Photontracer, Raytracer, TriangleTree, Hitpointmap
Chenhan Lyu	Scene, Photontracer, TriangleTree, Hitpointmap
Xiaoyu Liu (Mia)	Scene, Photontracer, Hitpointmap
Pipeline Design and Integration	
Yi Ge (Ellen)	Photon Mapping, Progressive Photon Mapping
Yuan Liu (Hive)	Photon Mapping, Progressive Photon Mapping
Rui Wang (Raymond)	Photon Mapping, Progressive Photon Mapping
Multithreading	
Yi Ge (Ellen)	Develop runnable version with multithreading under Windows
Yuan Liu (Hive)	Develop runnable version with multithreading under Windows
Rui Wang (Raymond)	Develop runnable version with multithreading under Windows
Testing and Verification	
Yi Ge (Ellen)	Complete multithreading test case: develop runnable version without multithreading under Linux and MacOS; develop runnable version with multithreading under Windows
Chenhan Lyu	Complete test cases for KD-Trees: test the correctness and robustness for two tree structure in codebase

8.2 Theory Research

Yi Ge (Ellen)	Photon Mapping, Progressive Photon Mapping
Yuan Liu (Hive)	Photon Mapping, Progressive Photon Mapping
Rui Wang (Raymond)	Photon Mapping

8.3 Report

Project Proposal	
Yi Ge (Ellen) Xiaoyu (Mia)	Abstract, Section 2, Section 4, Section 6, Appendix; Finalize the report Section 1, Section 3, Section 5
Progress Report	
Yi Ge (Ellen) Xiaoyu (Mia)	Section 1, Secion 3.4, Section 4, Section 6; Drew the Class Diagram as the Appendix; Finalize the report Section 2, Section 3, Section 5
Final Report	
Yi Ge (Ellen)	Design the whole structure and all contents of the report including: design the organization of the report/ design strategies for the testing evaluation (eg: test with and without multi-threading using different models) / design the experiment contents (eg: compare PPM under different iteration number, compare PM, PPM under different number of emitted photons, compare model under different scenes)
	Assign each part to each group mate and help them to revise the paper, adjust the structure and modify the content
	Help each group member to revise the paper
	Complete Section 3, Section 7, Section 8
	Run the experiments of RT, PM and PPM under Linux and MacOS(single thread)
Yuan Liu (Hive)	Run the experiments of RT, PM and PPM under Windows (multi threads)
	Complete Section 4
Rui Wang (Raymond)	Drew all the Class Diagram)
	Complete Section 3
Chenhan Lyu	Design and write the unit test, run the test for KD-tree under different operating systems
	Complete Section 5
	Format the whole paper
	Revise Section4,5 and 6
Xiaoyu Liu (Mia)	Complete abstract, Section 1, Section 2 and Section 9
	Help to wrote the design document and help Raymond to finish Section 4
	Help chenhan to format the whole paper

8.4 Other

Video making	Yuan Liu (Hive) and Yi Ge (Ellen)
Meeting Minute	Xiaoyu Liu (Mia), Chenhan Lyu
Codebase cleanup and final compeletion	Yi Ge (Ellen), Yuan Liu (Hive)

8.5 Meeting Attendence

Class Completion	
Yi Ge (Ellen)	5 times
Yuan Liu (Hive)	5 times
Rui Wang (Raymond)	3 times
Chenhan Lyu	4 times
Xiaoyu Liu	3 times

9 Conclusion and Acknowledgement

Approximately two months were spent working, the initial five weeks consisting of research, and the last two writing this thesis, producing more videos and tweaking the code where it was necessary, leaving around one week to implement the tool.

In general, this project was approached as a new learning experience, using the knowledge gained during the class to design an image rendering engine. The Results chapter, and videos that are handed along with this thesis, reveal that this tool was successful

The overall feeling, as the project reaches its end, is of success. Even though there are many topics not approached during this project, mainly due to the limited time and knowledge. However, this research experience makes us have a strong interest in computer graphics and hope to have a deeper understanding in the future

Our cooperation is an enjoyment. All members are actively involved in completing their tasks. We are very grateful to Prof. James O'Brien for his patient guidance. This research experience is very fruitful and significant for us. We believe that we will have more in-depth research on computer graphics in the future. We also look forward to working with Prof. James O'Brien on more topics in the future.

References

Jensen, H. W., Christensen, P. H. (1998, July). Efficient simulation of light transport in scenes with participating media using photon maps. In Proceedings of the 25th annual conference on Computer graphics and interactive techniques (pp. 311320).

Jensen, H. W. (1996, June). Global illumination using photon maps. In Eurographics workshop on Rendering techniques (pp. 21-30). Springer, Vienna.

Jensen, H. W. (2004). A practical guide to global illumination using ray tracing and photon mapping. In ACM SIGGRAPH 2004 Course Notes (pp. 20-es).

Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E.,

Purcell, T. J. (2007, March). A survey of general-purpose computation on graphics hardware. In Computer graphics forum (Vol. 26, No. 1, pp. 80-113). Oxford, UK: Blackwell Publishing Ltd.

Codebase We have referred to some Github open source projects, but we established our own codebase based on our pipeline design.
References include: https://github.com/ishaan13/PhotonMapper/tree/master/src
https://github.com/lazycal/Progressive-Photon-Mapping/blob/master/