

# Ray Tracing Renderer

Mei Yixuan

January 9, 2022

## 1 Introduction

This document describes the design of a simple ray tracing renderer, along with some sample scenes rendered using it. I finished it as course project for Advanced Computer Graphics, given by Prof. Shimming Hu in Tsinghua University.

Complete code for this renderer can be found in <https://github.com/AntonyMei/RayTracingRender>, along with all external libraries and resources. This code is designed for this course project only and implies no warranty, use at your own risk. All code follow Apache License 2.0, except those external libraries and resources.

## 2 Results

This section shows some of the demo scenes rendered using this renderer. All scenes are rendered under  $3840 * 2160$  (or  $3840 * 3840$  if aspect ratio is 1) with more than 1000 samples per pixel and tracing depth 50.

### 2.1 Hollow Glass Ball

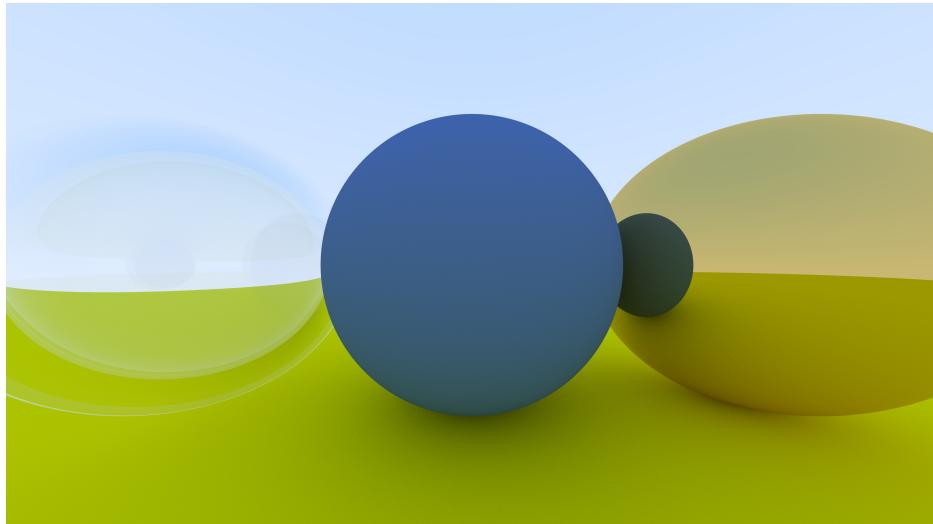


Figure 1: Hollow Glass Ball

This scene shows the usage of Lambertian (pure diffuse), Metallic (pure reflection) and Dielectric (refraction) material.

## 2.2 Hollow Glass Ball Small FOV

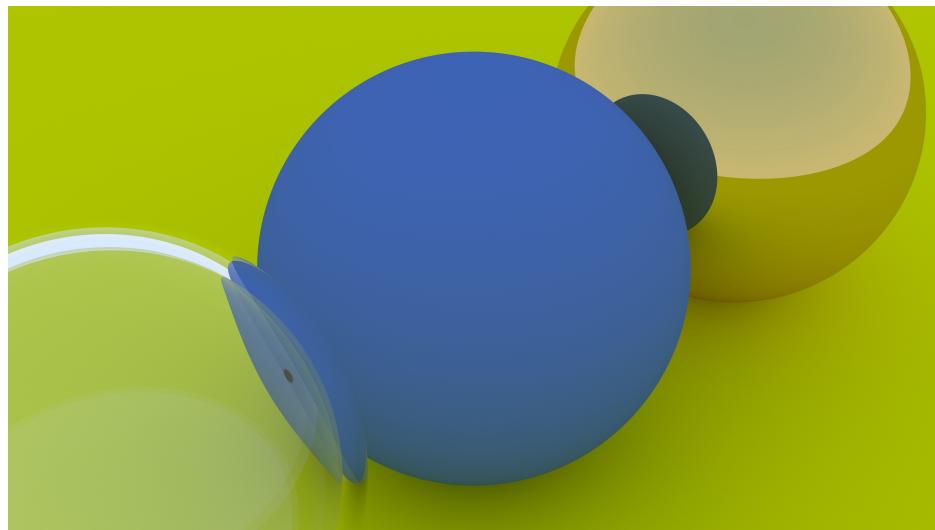


Figure 2: Hollow Glass Ball Small FOV

This scene is identical to the last one, but with smaller camera FOV. Note that edge of the dielectric ball has reflection rather than refraction, which is physically accurate.

## 2.3 Hollow Glass Ball Off Focus

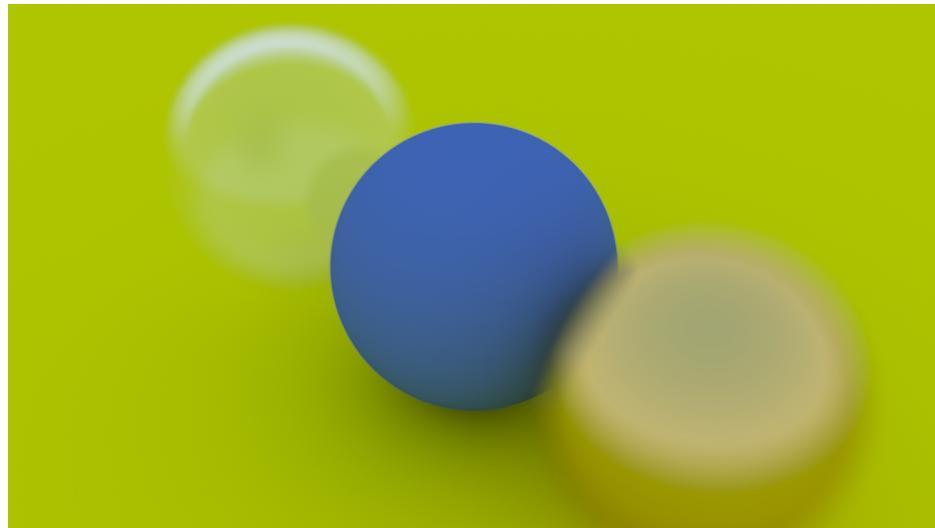


Figure 3: Hollow Glass Ball Off Focus

This scene shows off-focus blur effect.

## 2.4 Many Balls

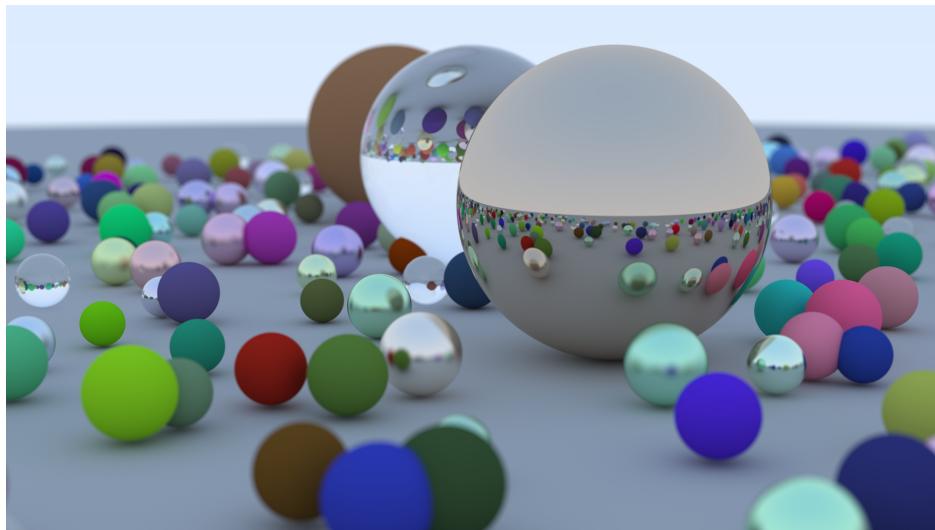


Figure 4: Many Balls

This scene contain many balls. Note that metallic material can use fuzziness to simulate imperfect reflection.

## 2.5 Motion Blur

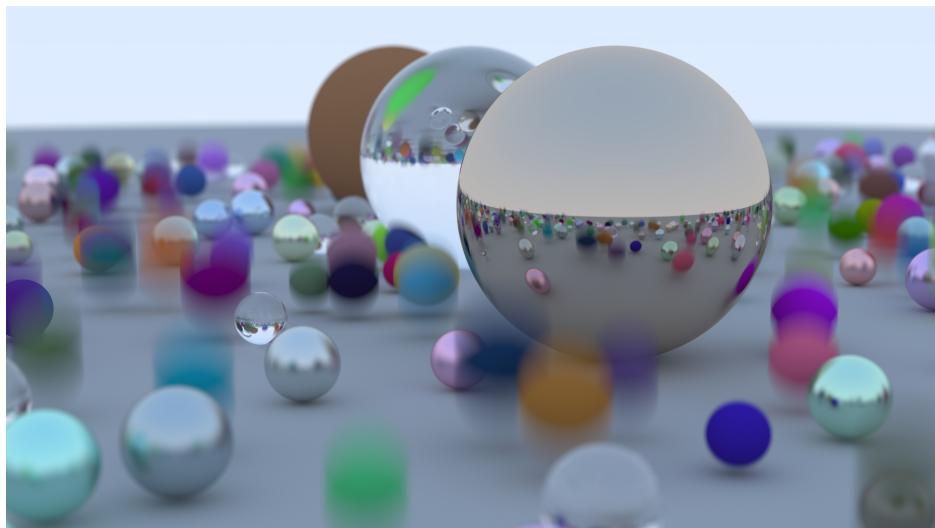


Figure 5: Motion Blur

This scene shows motion blur effect.

## 2.6 Motion Blur Checker

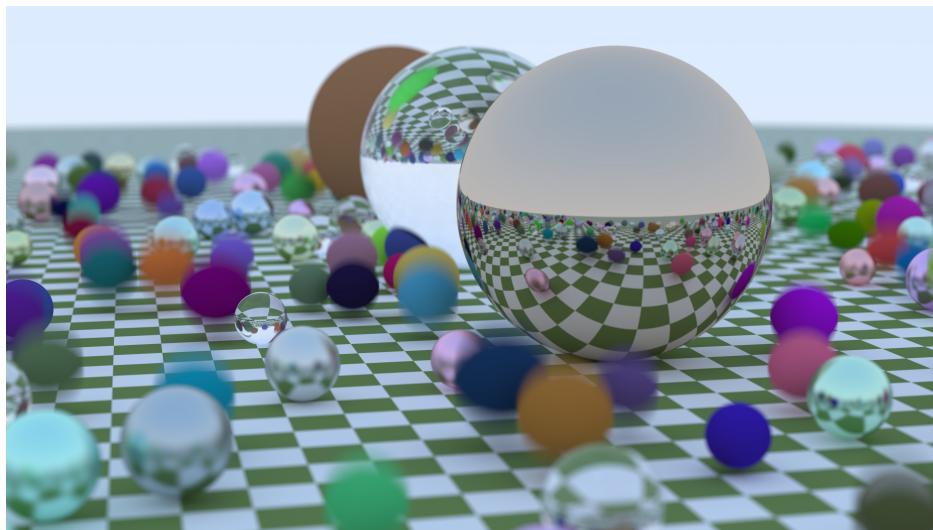


Figure 6: Motion Blur Checker

This scene adds a simple procedural texture (checker texture) to the last one.

## 2.7 Earth



Figure 7: Earth

This scene is a ball with image texture.

## 2.8 Cornell Box Series

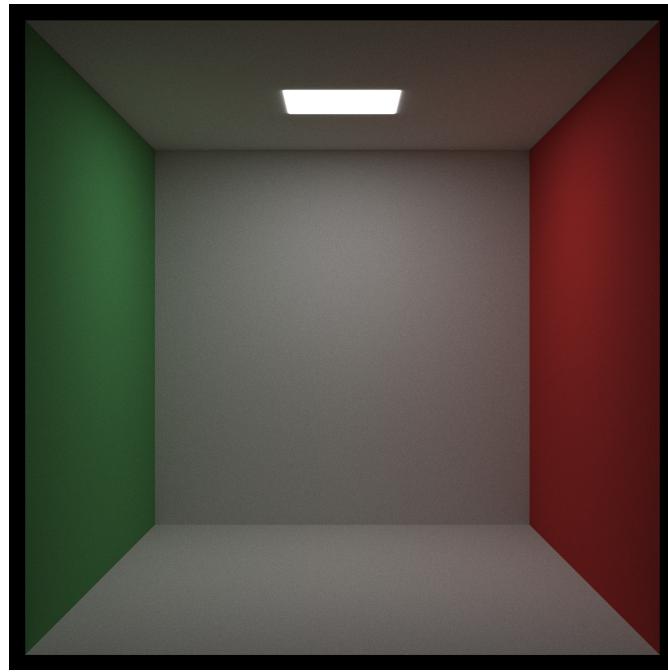


Figure 8: Cornell Box Empty

This is an empty Cornell Box constructed with axis-aligned rectangles.

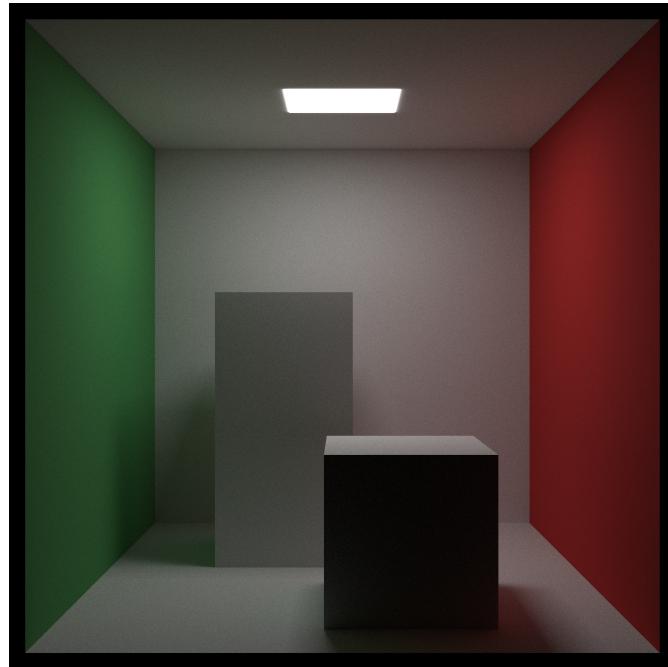


Figure 9: Cornell Box Tow Blocks

Added two boxes to the last scene.

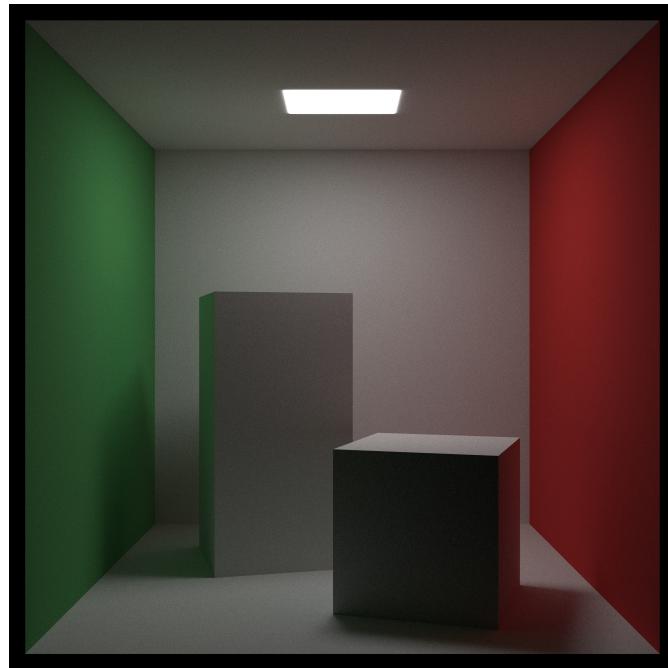


Figure 10: Cornell Box

Add rotation to the two boxes. Color bleeding is obvious on two surfaces facing walls.

## 2.9 Cornell Box Participating Media

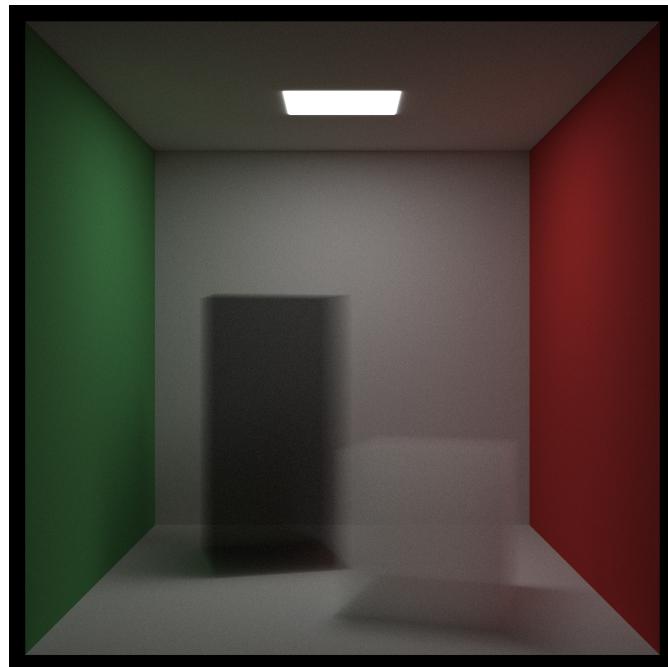


Figure 11: Cornell Box Participating Media

This scene replace the two original boxes with participating media boxes. These boxes are assigned with isotropic material to simulate the effect of smoke. This is a technique often used in volumetric rendering.

## 2.10 Book2 Final

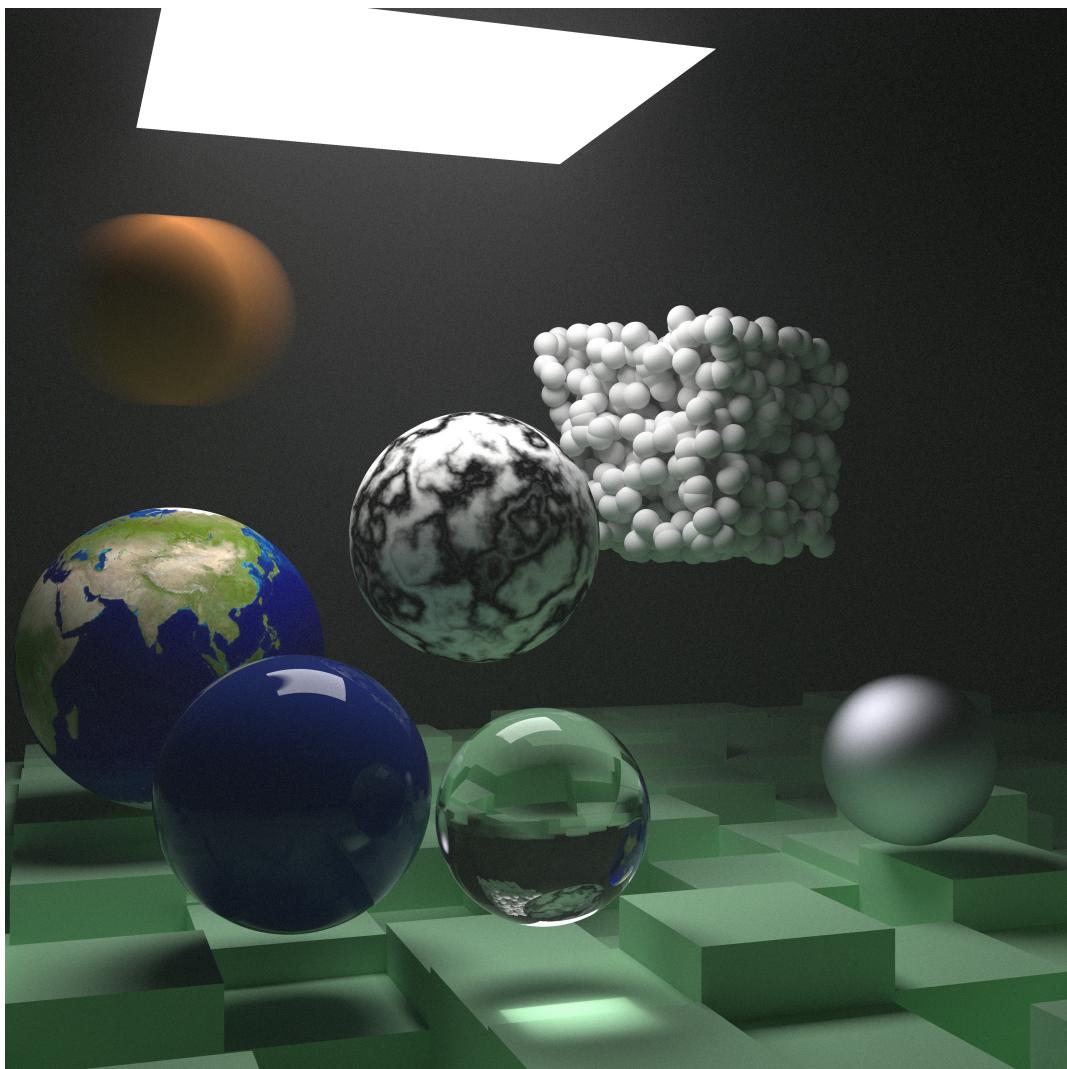


Figure 12: Book2 Final

This scene is a modified version of the one used by Peter Shirley in Ray Tracing Mini-Books 2. It combines many techniques. The whole scene is filled with thin smoke, rendered using participating media. The orange ball on the top left shows motion blur. The earth ball shows image textures. The marble ball in the middle is a complex example of procedural texture computed using Perlin noise. The box on the top right contains 10000 balls as components and uses BVH to accelerate.

## 2.11 Test Obj

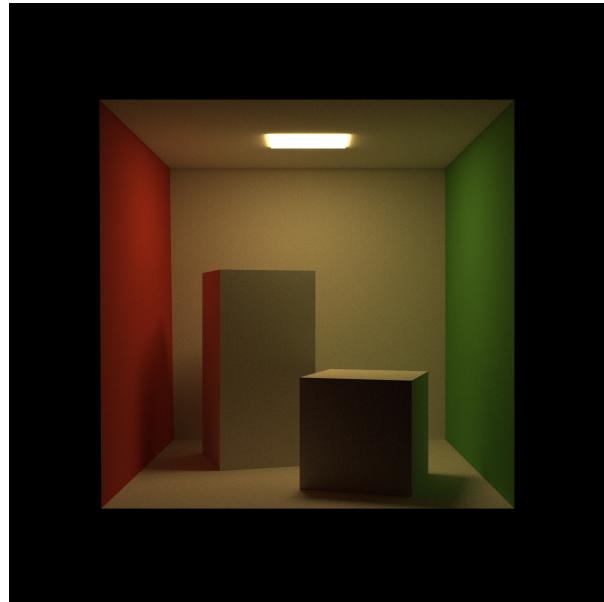


Figure 13: Test Obj

A simple scene used in obj test. This Cornell Box is made of triangles. It is stored in a .obj file (with material description in .mtl file) and imported using Tinyobjloader.

## 2.12 Sponza Sun



Figure 14: Sponza Sun

This scene is a more complicated one with image textures. Note that in this scene, a biased sampling technique is used such that the sun gets more samples.

## 2.13 Sponza Crytek Series



Figure 15: Sponza Crytek Cloudy

This scene is a remastered version of the last one from Crytek cooperation. It is rendered unbiasedly with Path Tracing. Note that bump map is enabled to add more details to the geometry (like the lion in the back). The following one is the same scene with a more complicated skybox. It can be used to simulate extremely sunny weather.

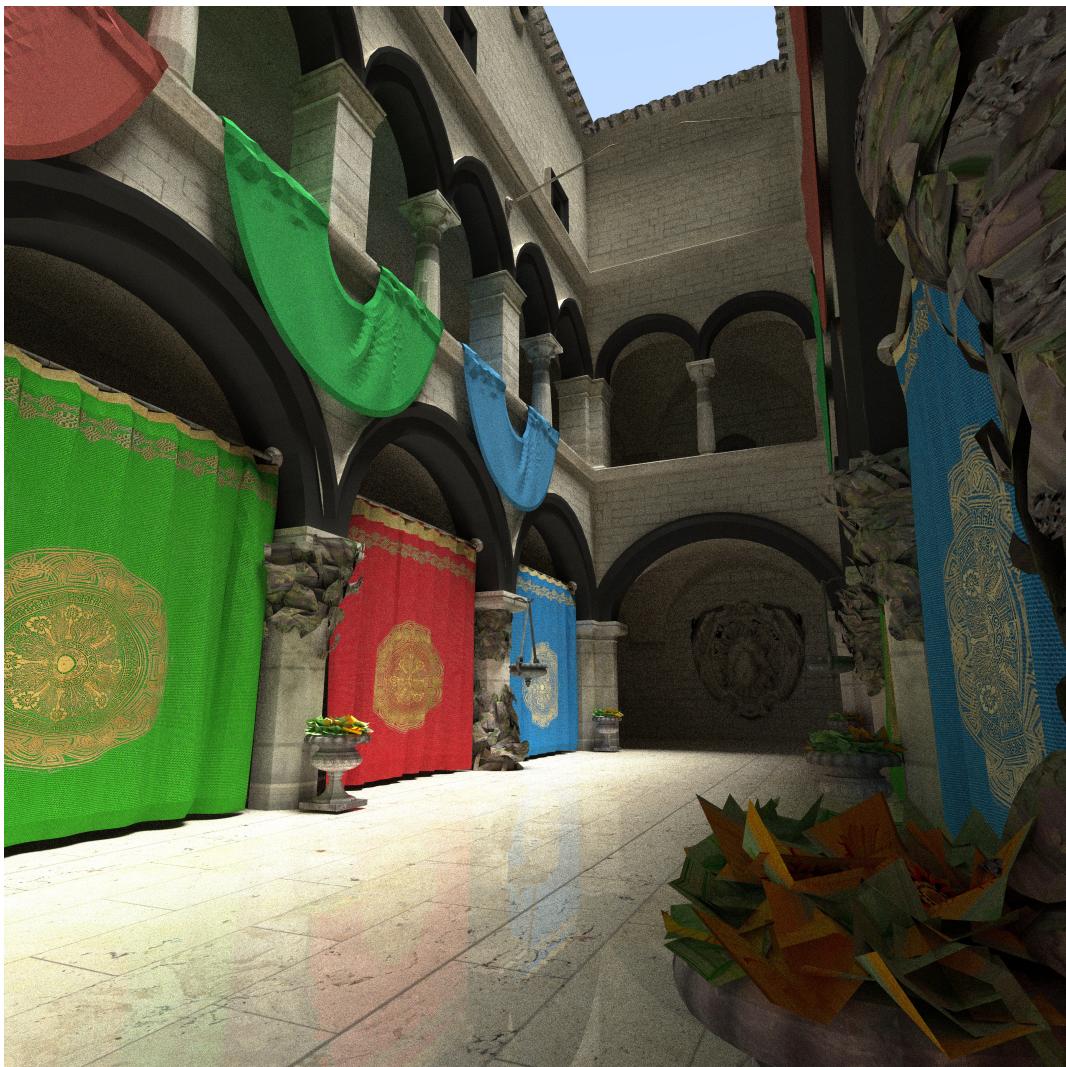


Figure 16: Sponza Crytek Sunny

## 2.14 Photon Mapping Series

A bunny rendered using Photon Mapping. When rendering the first image, photon tracing depth is set to one so that the caustic effect can be seen clearly. The correct effect is shown in the second image.

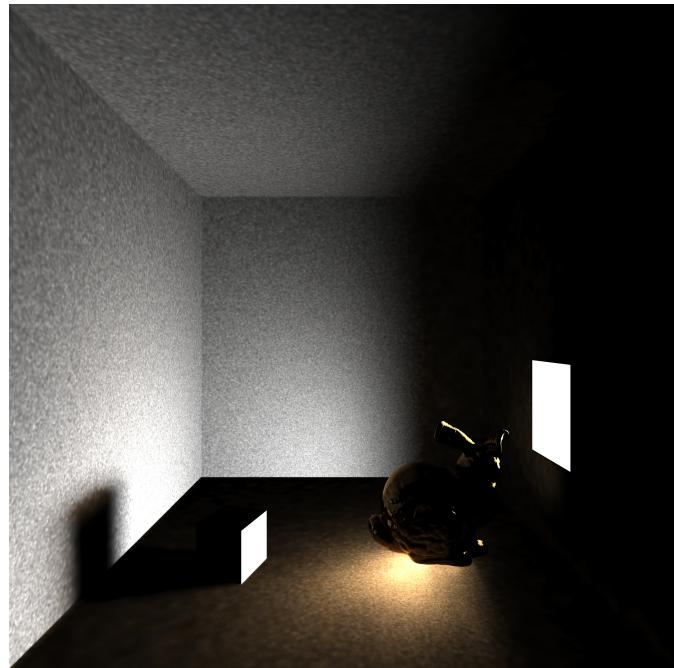


Figure 17: Photon Mapping Bunny Caustic

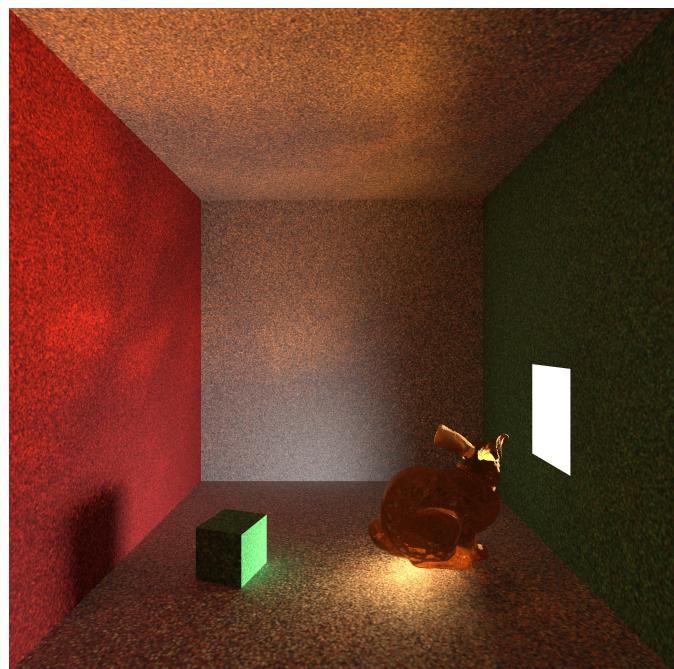


Figure 18: Photon Mapping Bunny

## 3 Usage

### How set scene before compilation

**step 1:** set image settings (customization (e.g. changing aspect\_ratio to 16.0 / 9.0) can be done by modifying render\_scene in main.cpp)

**step 2:** choose integrator (Photon Mapping only supports one scene, Path Tracing has step 3 of setting scene)

**step 3:** set scene, camera and skybox (scene functions can be found in src/scenes)

### How to run this renderer

#### Windows:

**step 1:** compile with CLion using MSVC compiler

**step 2:** run "Project.exe 8", here 8 means using 8 threads (after running we will get 8 .partial files)

**step 3:** run "packager.exe 8", here 8 means combining 8 partial files (after running we will get a .ppm image, which can be opened using OpenSeeIt)

**step 4:** run python convert.py, which converts .ppm into .jpg (note that opencv for python is required to run this)

#### Linux:

run "bash linux\_run.sh", which compiles the renderer, uses 80 processes to run it and process result into a .ppm and a .jpg file. (Note that for default Sponza Crytek scene, about 150 GB memory is required. This requirement is proportional to the number of processes.)

## 4 Renderer Function Menu

The basic functionalities of this renderer are listed in this section. Detailed description with code and explanation can be found in the following chapters.

### 4.1 Core Renderer:

**Integrator:** Monte Carlo Path Tracing, Photon Mapping.

**Accelerator:** BVH (AABB with SAH), Kd-Tree.

**Hardware Acceleration:** OpenMP on Windows, Multi-processing on Linux.

### 4.2 Objects:

**Hittable Objects:** Triangle, Triangular Mesh, Box, Sphere.

**Complex Models:** .obj Model with .mtl Material Description.

**Skybox:** Constant Skybox, Directional Skybox, Realistic Skybox, Multi-layer Skybox.

**Transforms:** Rotation, Translation.

### 4.3 Materials & Textures

**Materials:** Lambertian, Metal, Dielectric, PBR material, Isotropic, Diffuse Light.

**Textures:** Color Texture, Checker Texture, Perlin Noise Texture, Marble Texture, Image Texture, Bump Texture.

### 4.4 Visual Effects

**Camera Effects:** Off-focus Blur, Motion Blur.

**Volumetric Rendering:** Participating Media.

**Other Visual Effects:** Gamma Correction, Caustic.

## 5 Renderer Infrastructures

### 5.1 Math Utilities

#### 5.1.1 Basic Math Utilities (/src/math/Utils.h)

```
1 //  
2 // Created by meiyixuan on 2021-12-10.  
3 //  
4  
5 #ifndef PROJECT_UTILS_H  
6 #define PROJECT_UTILS_H  
7  
8 // path  
9 inline std::string model_pth() {  
10    #if defined(WINDOWS)  
11        return "./";  
12    #else  
13        return "./resources/";  
14    #endif  
15 }  
16  
17 // memory  
18 using std::shared_ptr;  
19 using std::make_shared;  
20  
21 // statistics  
22 int normal_triangles = 0;  
23 int u_degrade_triangles = 0;  
24 int v_degrade_triangles = 0;  
25 int unrecoverable_triangles = 0;  
26  
27 // integrator  
28 int integrator_type;  
29  
30 int use_photon_map() { return 1; }  
31  
32 int use_path_tracing() { return 0; }  
33  
34 // math  
35 using std::sqrt;  
36 const double inf = std::numeric_limits<double>::infinity();  
37 const double TMIN = 0.001;  
38 const double EPSILON = 0.0000001;  
39 const double pi = 3.1415926535897932385;  
40  
41 inline double deg2rad(double deg) {  
42    return deg * pi / 180.0;  
43 }  
44  
45 inline double clamp(double x, double min, double max) {  
46    if (x < min) return min;  
47    if (x > max) return max;  
48    return x;
```

```

49 }
50
51 // random
52 inline double random_double_fixed() {
53     // random number in [0, 1), with fixed initial seed
54     // This function is thread-identical, and should be used in
55     // perlin noise and random scene generation.
56     return rand() / (RAND_MAX + 1.0);
57 }
58
59 inline double random_double() {
60     if (integrator_type == 1) {
61         return random_double_fixed();
62     }
63     // random number in [0, 1)
64     static std::random_device rd;
65     static std::uniform_real_distribution<double> distribution(0.0, 1.0);
66     static std::mt19937 generator(rd());
67     return distribution(generator);
68 }
69
70 inline double random_double(double min, double max) {
71     // random real in [min,max].
72     return min + (max - min) * random_double();
73 }
74
75 inline double random_double_fixed(double min, double max) {
76     // random real in [min,max].
77     return min + (max - min) * random_double_fixed();
78 }
79
80 inline int random_int_fixed(int min, int max) {
81     // Returns a random integer in [min,max].
82     return static_cast<int>(random_double_fixed(min, max + 1));
83 }
84
85 // input
86 int parse_int(const char *buf, size_t len) {
87     int res = 0;
88     for (size_t i = 0; i < len; ++i) {
89         res *= 10;
90         res += buf[i] - '0';
91     }
92     return res;
93 }
94
95 #endif //PROJECT_UTILS_H

```

Basic math and random utilities. Note that we need a predictable random function (`random_double_fixed`) for multiproCESSing, since some random number sequences must be identical across processes (e.g. random numbers used in scene initialization).

### 5.1.2 Vector Math (/src/math/Vector3d.h)

```

1 //
2 // Created by meiyixuan on 2021-12-09.
3 // Basic math class for vector operations.

```

```

4 // 
5 
6 #ifndef PROJECT_VECTOR3D_H
7 #define PROJECT_VECTOR3D_H
8 
9 class Vector3d {
10 public:
11 // constructors
12 Vector3d() : val{0, 0, 0} {}

13 Vector3d(double x, double y, double z) : val{x, y, z} {}

14 // data
15 double x() const { return val[0]; }

16 double y() const { return val[1]; }

17 double z() const { return val[2]; }

18 // operators
19 Vector3d &operator+=(const Vector3d &v) {
20     val[0] += v.val[0];
21     val[1] += v.val[1];
22     val[2] += v.val[2];
23     return *this;
24 }
25 
26 Vector3d &operator-=(const Vector3d &v) {
27     val[0] -= v.val[0];
28     val[1] -= v.val[1];
29     val[2] -= v.val[2];
30     return *this;
31 }
32 
33 Vector3d &operator*=(const double t) {
34     val[0] *= t;
35     val[1] *= t;
36     val[2] *= t;
37     return *this;
38 }
39 
40 Vector3d &operator/=(const double t) {
41     return *this *= 1 / t;
42 }
43 
44 Vector3d operator-() const { return {-val[0], -val[1], -val[2]}; }

45 double operator[](int i) const { return val[i]; }

46 double &operator[](int i) { return val[i]; }

47 double length() const {
48     return std::sqrt(squared_length());
49 }
50 
51 double squared_length() const {
52     return val[0] * val[0] + val[1] * val[1] + val[2] * val[2];
53 }
54 
```

```

61 }
62
63 // utils
64 inline static Vector3d random() {
65     return {random_double(), random_double(), random_double()};
66 }
67
68 inline static Vector3d random_fixed() {
69     return {random_double_fixed(), random_double_fixed(), random_double_fixed()};
70 }
71
72 inline static Vector3d random(double min, double max) {
73     return {random_double(min, max), random_double(min, max), random_double(min, max)};
74 }
75
76 inline static Vector3d random_fixed(double min, double max) {
77     return {random_double_fixed(min, max), random_double_fixed(min, max), random_double_fixed(min, max)};
78 }
79
80 bool near_zero() const {
81     // Return true if the vector is close to zero in all dimensions.
82     const auto s = 1e-8;
83     return (fabs(val[0]) < s) && (fabs(val[1]) < s) && (fabs(val[2]) < s);
84 }
85
86 public:
87
88     double val[3];
89
90     // for access to val directly
91     friend inline std::ostream &operator<<(std::ostream &out, const Vector3d &v);
92
93     friend inline Vector3d operator+(const Vector3d &u, const Vector3d &v);
94
95     friend inline Vector3d operator-(const Vector3d &u, const Vector3d &v);
96
97     friend inline Vector3d operator*(const Vector3d &u, const Vector3d &v);
98
99     friend inline Vector3d operator*(double t, const Vector3d &v);
100
101    friend inline double dot(const Vector3d &u, const Vector3d &v);
102
103    friend inline Vector3d cross(const Vector3d &u, const Vector3d &v);
104};

105
106 using Point = Vector3d;    // 3D point
107 using Color = Vector3d;    // RGB color
108
109 inline std::ostream &operator<<(std::ostream &out, const Vector3d &v) {
110     return out << v.val[0] << ' ' << v.val[1] << ' ' << v.val[2];
111 }
112
113 inline Vector3d operator+(const Vector3d &u, const Vector3d &v) {
114     return {u.val[0] + v.val[0], u.val[1] + v.val[1], u.val[2] + v.val[2]};
115 }
116

```

```

117 inline Vector3d operator-(const Vector3d &u, const Vector3d &v) {
118     return {u.val[0] - v.val[0], u.val[1] - v.val[1], u.val[2] - v.val[2]};
119 }
120
121 inline Vector3d operator*(const Vector3d &u, const Vector3d &v) {
122     return {u.val[0] * v.val[0], u.val[1] * v.val[1], u.val[2] * v.val[2]};
123 }
124
125 inline Vector3d operator*(double t, const Vector3d &v) {
126     return {t * v.val[0], t * v.val[1], t * v.val[2]};
127 }
128
129 inline Vector3d operator*(const Vector3d &v, double t) {
130     return t * v;
131 }
132
133 inline Vector3d operator/(Vector3d v, double t) {
134     return (1 / t) * v;
135 }
136
137 inline double dot(const Vector3d &u, const Vector3d &v) {
138     return u.val[0] * v.val[0] + u.val[1] * v.val[1] + u.val[2] * v.val[2];
139 }
140
141 inline Vector3d cross(const Vector3d &u, const Vector3d &v) {
142     return {u.val[1] * v.val[2] - u.val[2] * v.val[1],
143             u.val[2] * v.val[0] - u.val[0] * v.val[2],
144             u.val[0] * v.val[1] - u.val[1] * v.val[0]};
145 }
146
147 inline Vector3d normalize(Vector3d v) {
148     return v / v.length();
149 }
150
151 Vector3d random_in_unit_disk() {
152     while (true) {
153         auto p = Vector3d(random_double(-1, 1), random_double(-1, 1), 0);
154         if (p.squared_length() >= 1) continue;
155         return p;
156     }
157 }
158
159 Vector3d random_in_unit_sphere() {
160     // uses reject sampling to generate a random point
161     while (true) {
162         auto p = Vector3d::random(-1, 1);
163         if (p.squared_length() >= 1) continue;
164         return p;
165     }
166 }
167
168 Vector3d random_unit_vector() {
169     return normalize(random_in_unit_sphere());
170 }
171
172 Vector3d random_in_hemisphere(const Vector3d &normal) {
173     Vector3d in_unit_sphere = random_in_unit_sphere();

```

```

174     if (dot(in_unit_sphere, normal) > 0.0)
175         return in_unit_sphere;
176     else
177         return -in_unit_sphere;
178     }
179
180     Vector3d reflect(const Vector3d &v, const Vector3d &n) {
181         return v - 2 * dot(v, n) * n;
182     }
183
184     Vector3d refract(const Vector3d &uv, const Vector3d &n, double refract_coefficient) {
185         // eta: typically air = 1.0, glass = 1.3 – 1.7, diamond = 2.4
186         // refract_coefficient is equal to eta_in / eta_out
187         auto cos_theta = fmin(dot(-uv, n), 1.0);
188         Vector3d r_out_perp = refract_coefficient * (uv + cos_theta * n);
189         Vector3d r_out_parallel = -sqrt(fabs(1.0 - r_out_perp.squared_length())) * n;
190         return r_out_perp + r_out_parallel;
191     }
192
193 #endif //PROJECT_VECTOR3D_H

```

This file contains all we need for vector math. Note that reflection, refraction and random vector sampling are all defined here as well.

### 5.1.3 Perlin Noise (/src/math/Perlin.h)

```

1 //
2 // Created by meiyixuan on 2021-12-18.
3 // Perlin noise generator uses code from Peter Shirley.
4 // Warning: Perlin noise generator must have identical seeds across all processes.
5 //
6
7 #ifndef PROJECT_PERLIN_H
8 #define PROJECT_PERLIN_H
9
10 class PerlinNoise {
11     public:
12     PerlinNoise() {
13         rand_vector = new Vector3d[point_count];
14         for (int i = 0; i < point_count; ++i) {
15             rand_vector[i] = normalize(Vector3d::random_fixed(-1, 1));
16         }
17
18         perm_x = perlin_generate_perm();
19         perm_y = perlin_generate_perm();
20         perm_z = perlin_generate_perm();
21     }
22
23     ~PerlinNoise() {
24         delete [] rand_vector;
25         delete [] perm_x;
26         delete [] perm_y;
27         delete [] perm_z;
28     }
29
30     double turbulence(const Point &p, int sample_depth = 7) const {
31         // initialize

```

```

32     auto sum = 0.0;
33     auto cur_point = p;
34     auto weight = 1.0;
35
36     // weighted sum of samples
37     for (int i = 0; i < sample_depth; i++) {
38         sum += weight * noise(cur_point);
39         weight *= 0.5;
40         cur_point *= 2;
41     }
42     return fabs(sum);
43 }
44
45 double noise(const Point &p) const {
46     // only one sample from perlin noise
47     auto u = p.x() - floor(p.x());
48     auto v = p.y() - floor(p.y());
49     auto w = p.z() - floor(p.z());
50     auto i = static_cast<int>(floor(p.x()));
51     auto j = static_cast<int>(floor(p.y()));
52     auto k = static_cast<int>(floor(p.z()));
53     Vector3d c[2][2][2];
54
55     for (int di = 0; di < 2; di++)
56         for (int dj = 0; dj < 2; dj++)
57             for (int dk = 0; dk < 2; dk++)
58                 c[di][dj][dk] = rand_vector[
59                     perm_x[(i + di) & 255] ^
60                     perm_y[(j + dj) & 255] ^
61                     perm_z[(k + dk) & 255]];
62
63     // tri-linear interpolation
64     return perlin_interpolation(c, u, v, w);
65 }
66
67 private:
68     static const int point_count = 256;
69     Vector3d *rand_vector;
70     int *perm_x;
71     int *perm_y;
72     int *perm_z;
73
74     static int *perlin_generate_perm() {
75         auto p = new int[point_count];
76         for (int i = 0; i < PerlinNoise::point_count; i++)
77             p[i] = i;
78         permute(p, point_count);
79         return p;
80     }
81
82     static void permute(int *p, int n) {
83         for (int i = n - 1; i > 0; i--) {
84             int target = random_int_fixed(0, i);
85             int tmp = p[i];
86             p[i] = p[target];
87             p[target] = tmp;
88         }

```

```

89 }
90
91 static double perlin_interpolation(Vector3d c[2][2][2], double u, double v, double w) {
92     // hermite smoothing to eliminate Mach Band
93     auto uu = u * u * (3 - 2 * u);
94     auto vv = v * v * (3 - 2 * v);
95     auto ww = w * w * (3 - 2 * w);
96
97     // interpolate
98     auto accum = 0.0;
99     for (int i = 0; i < 2; i++)
100    for (int j = 0; j < 2; j++)
101    for (int k = 0; k < 2; k++) {
102        Vector3d weight_v(u - i, v - j, w - k);
103        accum += (i * uu + (1 - i) * (1 - uu))
104            * (j * vv + (1 - j) * (1 - vv))
105            * (k * ww + (1 - k) * (1 - ww))
106            * dot(c[i][j][k], weight_v);
107    }
108    return accum;
109 }
110 };
111
112 #endif //PROJECT_PERLIN_H

```

This file defines a simple Perlin Noise Generator. Perlin Noise is a technique often used in procedural texture generation. In this project, it is used to generate marble texture.

## 5.2 Basic Datatypes

### 5.2.1 Ray (src/core/Ray.h)

```

1 /**
2 // Created by meiyixuan on 2021-12-09.
3 // This file contains the definition of Ray.
4 /**
5
6 #ifndef PROJECT_RAY_H
7 #define PROJECT_RAY_H
8
9 class Ray {
10 public:
11     Ray() : o(), d(), tm{} {}
12
13     Ray(const Point &origin, const Vector3d &direction, double time = 0.0, bool _is_camera_ray = false) :
14     o(origin), d(direction), tm(time), is_camera_ray(_is_camera_ray) {}
15
16     // utils
17     Point origin() const { return o; }
18
19     Vector3d direction() const { return d; }
20
21     double time() const { return tm; }
22
23     bool camera_ray() const { return is_camera_ray; }
24
25     Point at(double t) const { return o + t * d; }

```

```

26
27     private:
28     // origin and direction
29     Point o;
30     Vector3d d;
31     double tm;
32     bool is_camera_ray{ false };
33 };
34
35
36 #endif //PROJECT_RAY_H

```

This file defines Ray. Each ray has an origin, a direction and a timestamp. The timestamp is used in motion blur. Also, *is\_camera\_ray* here is used in skybox to separate visual effect from real lighting effect.

### 5.2.2 Pixel (src/core/Pixel.h)

```

1 //
2 // Created by meiyixuan on 2021-12-09.
3 // This file contains the definition of Pixel.
4 //
5
6 #ifndef PROJECT_PIXEL_H
7 #define PROJECT_PIXEL_H
8
9 class Pixel {
10 public:
11     Pixel() : pixel_color(), write_flag(false), sample_count(1) {}
12
13     void set(const Color &color, int samples_per_pixel) {
14         pixel_color = color;
15         sample_count = samples_per_pixel;
16     }
17
18     Color get_color() const { return pixel_color; }
19
20     int get_sample_count() const { return sample_count; }
21
22     void write(std::ofstream &out) {
23         // a pixel can only be written once for correctness
24         if (!write_flag) {
25             write_flag = true;
26         } else {
27             throw std::runtime_error("A pixel is written twice");
28         }
29
30         // divide by sample count and gamma correct
31         auto r = pixel_color.x();
32         auto g = pixel_color.y();
33         auto b = pixel_color.z();
34         auto scale = 1.0 / sample_count;
35         r = sqrt(scale * r);
36         g = sqrt(scale * g);
37         b = sqrt(scale * b);
38
39         // scale to [0, 255] and output
40         out << static_cast<int>(256 * clamp(r, 0.0, 0.999)) << ', '

```

```

41     << static_cast<int>(256 * clamp(g, 0.0, 0.999)) << ', '
42     << static_cast<int>(256 * clamp(b, 0.0, 0.999)) << '\n';
43 }
44
45 private:
46     bool write_flag;
47     Color pixel_color;
48     int sample_count;
49 };
50
51 #endif //PROJECT_PIXEL_H

```

This file defines Pixel, a data structure that is responsible for writing pixel into files. Note that in line 30 - 37 we add a simple Gamma correction to make the image more realistic.

### 5.2.3 Hit (src/core/Hit.h)

```

1 //
2 // Created by meiyixuan on 2021-12-18.
3 //
4
5 #ifndef PROJECT_HIT_H
6 #define PROJECT_HIT_H
7
8 struct Hit {
9     Point hit_point;
10    Vector3d normal;
11    std::shared_ptr<Material> mat_ptr;
12    double t{0};
13    double u{0};
14    double v{0};
15    bool front_face{false};
16    // 0: diffuse, 1: reflect, 2: refract, 3: isotropic
17    int scatter_mode{-1};
18    int remaining_bounce{-1};
19
20    inline void set_face_normal(const Ray &r, const Vector3d &n) {
21        front_face = (dot(r.direction(), n) < 0);
22        normal = front_face ? n : -n;
23    }
24};
25
26#endif //PROJECT_HIT_H

```

This data structure contains everything needed to be stored when a hit happens.

### 5.2.4 AABB (src/core/AABB.h)

```

1 //
2 // Created by meiyixuan on 2021-12-15.
3 //
4
5 #ifndef PROJECT_AABB_H
6 #define PROJECT_AABB_H
7
8 class AABB {
9     public:

```

```

10    AABB() = default;
11
12    AABB(const Point &min, const Point &max) : minimum(min), maximum(max) {}
13
14    Point min() const { return minimum; }
15
16    Point max() const { return maximum; }
17
18    inline bool hit(const Ray &ray, double t_min, double t_max) const {
19        // an optimized slab method (Andrew Kensler at Pixar)
20        for (int a = 0; a < 3; a++) {
21            auto invD = 1.0f / ray.direction()[a];
22            auto t0 = (min()[a] - ray.origin()[a]) * invD;
23            auto t1 = (max()[a] - ray.origin()[a]) * invD;
24            if (invD < 0.0f)
25                std::swap(t0, t1);
26            t_min = t0 > t_min ? t0 : t_min;
27            t_max = t1 < t_max ? t1 : t_max;
28            if (t_max <= t_min)
29                return false;
30        }
31        return true;
32    }
33
34    inline int longest_axis() const {
35        Vector3d length = maximum - minimum;
36        if (length[0] >= length[1] && length[0] >= length[2])
37            return 0;
38        else if (length[1] >= length[0] && length[1] >= length[2])
39            return 1;
40        else
41            return 2;
42    }
43
44    inline double area() const {
45        Vector3d length = maximum - minimum;
46        return 2 * (length[0] * length[1] +
47                    length[0] * length[2] +
48                    length[1] * length[2]);
49    }
50
51    private:
52        // ranges in three directions (i.e. two points in diagonal)
53        Point minimum;
54        Point maximum;
55    };
56
57    AABB surrounding_box(AABB box0, AABB box1) {
58        Point min(fmin(box0.min().x(), box1.min().x()),
59                  fmin(box0.min().y(), box1.min().y()),
60                  fmin(box0.min().z(), box1.min().z()));
61
62        Point max(fmax(box0.max().x(), box1.max().x()),
63                  fmax(box0.max().y(), box1.max().y()),
64                  fmax(box0.max().z(), box1.max().z()));
65
66        return {min, max};

```

```

67 }
68
69 #endif //PROJECT_AABB_H

```

AABB is abbreviation of axis-aligned-bounding-box, which is used in BVH. Here, *area()* and *longest\_axis()* are used in SAH (surface area heuristic) to make the BVH more balanced spatially.

### 5.2.5 Photon (src/core/Photon.h)

```

1 //
2 // Created by meiyixuan on 2021-12-29.
3 //
4
5 #ifndef PROJECT_PHOTON_H
6 #define PROJECT_PHOTON_H
7
8 struct Photon {
9     public:
10    Photon() = default;
11
12    Photon(Vector3d pos, Vector3d dir, Vector3d _power, int _axis = 0)
13        : position(pos), direction(dir), power(_power), axis(_axis) {}
14
15    public:
16    Vector3d position;
17    Vector3d direction;
18    Vector3d power; // in color
19    int axis{0};
20};
21
22#endif //PROJECT_PHOTON_H

```

This file contains definition of Photon. A photon stores light intensity at given position. It is generated in forward pass (photon map generation) of Photon Mapping and used in backward pass.

### 5.2.6 NearestPhotons (src/core/NearestPhotons.h)

```

1 //
2 // Created by meiyixuan on 2021-12-29.
3 //
4
5 #ifndef PROJECT_NEARESTPHOTONS_H
6 #define PROJECT_NEARESTPHOTONS_H
7
8 struct NearestPhotons {
9     public:
10    NearestPhotons() {
11        max_photons = 0;
12        found_photons = 0;
13        heap_full = false;
14        dist_square = nullptr;
15        photons = nullptr;
16    }
17
18    ~NearestPhotons() {
19        delete [] dist_square;

```

```

20     delete [] photons;
21 }
22
23 public:
24 Vector3d pos;
25 int max_photons, found_photons;
26 bool heap_full;
27 double* dist_square;
28 Photon** photons;
29 };
30
31 #endif //PROJECT_NEARESTPHOTONS_H

```

NearestPhotons is a data structure that store nearest photons at given position. It is generated from full PhotonMap and used to determine light intensity at a given point on diffuse surface.

### 5.3 External Libraries

## 6 References

#### Code References:

- [1] Ray Tracing Mini-books, by Peter Shirley (<https://raytracing.github.io>)
- [2] Physically Based Rendering: From Theory to Implementation, by Matt Pharr, Wenzel Jakob, and Greg Humphreys (<https://github.com/mmp/pbrt-v3>)
- [3] Dezeming Family (<https://dezeming.top/>)

#### External Libraries:

- [1] Tinyobjloader (<https://github.com/tinyobjloader/tinyobjloader>)
- [2] stb\_image (<https://github.com/nothings/stb>)

#### Model Resources:

- [1] Morgan McGuire, Computer Graphics Archive, July 2017 (<https://casual-effects.com/data>)