

Ray Tracing Renderer

Mei Yixuan

January 11, 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

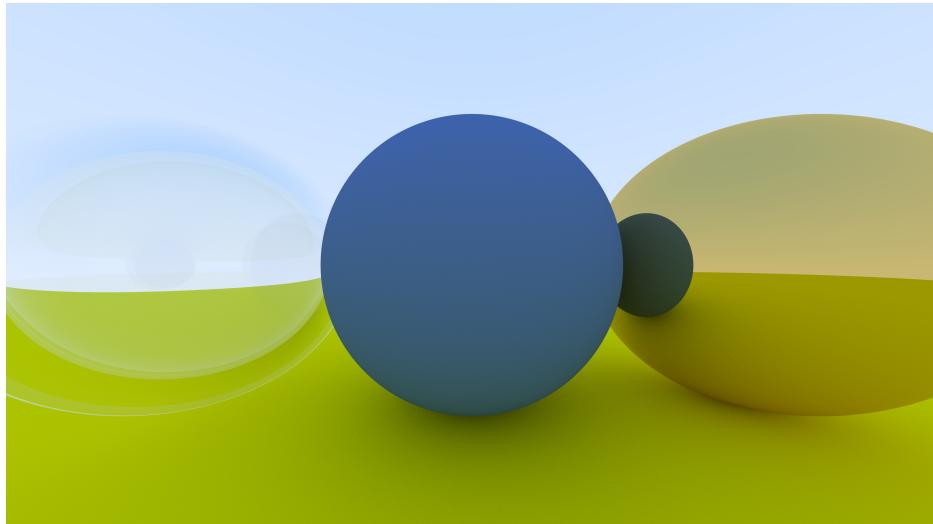


Figure 1: Hollow Glass Ball

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

2.2 Hollow Glass Ball Small FOV

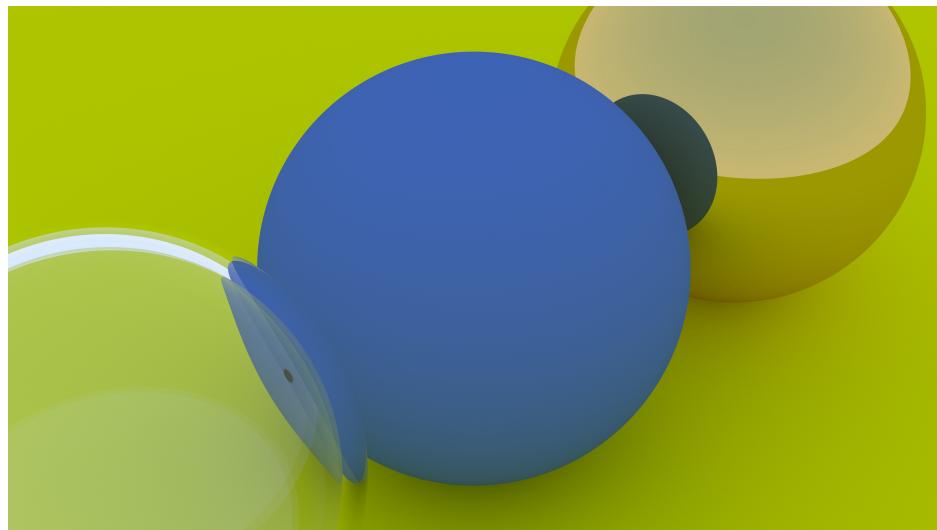


Figure 2: Hollow Glass Ball Small FOV

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

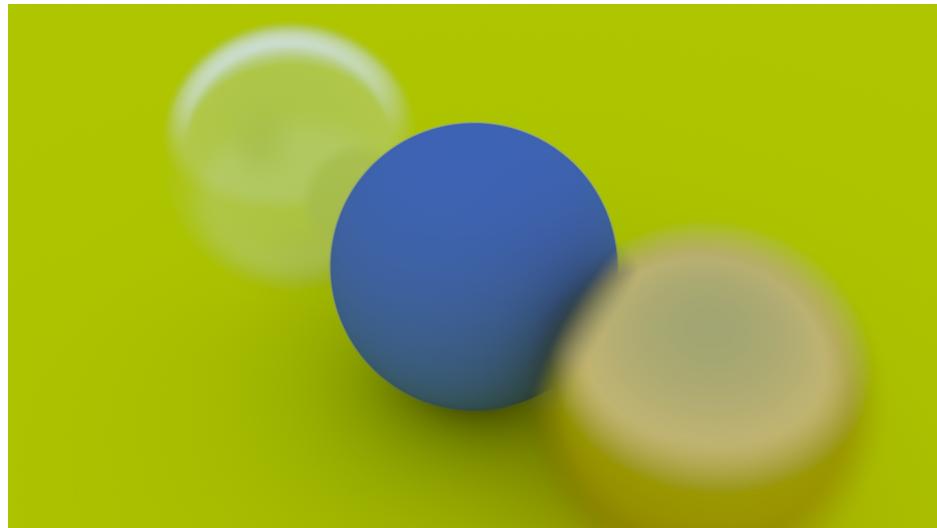


Figure 3: Hollow Glass Ball Off Focus

This scene shows off-focus blur effect.

2.4 Many Balls

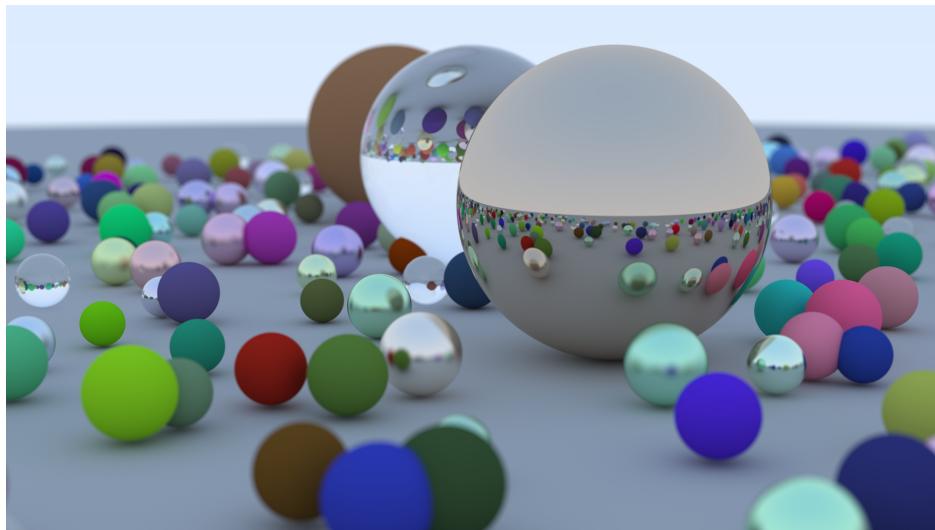


Figure 4: Many Balls

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

2.5 Motion Blur

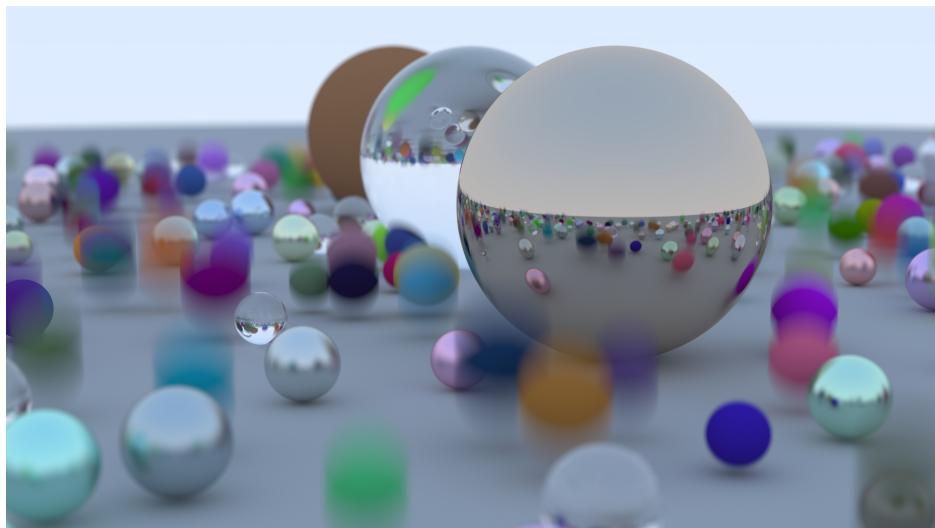


Figure 5: Motion Blur

This scene shows motion blur effect.

2.6 Motion Blur Checker

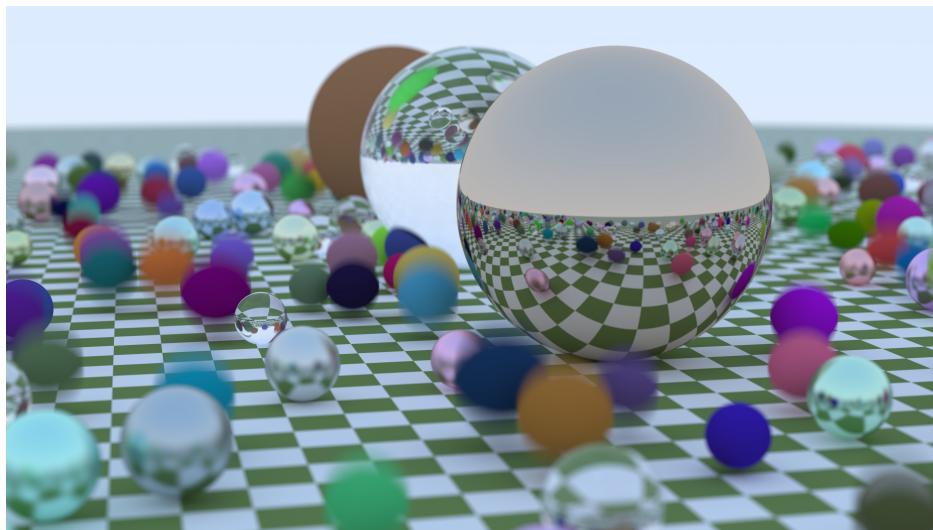


Figure 6: Motion Blur Checker

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

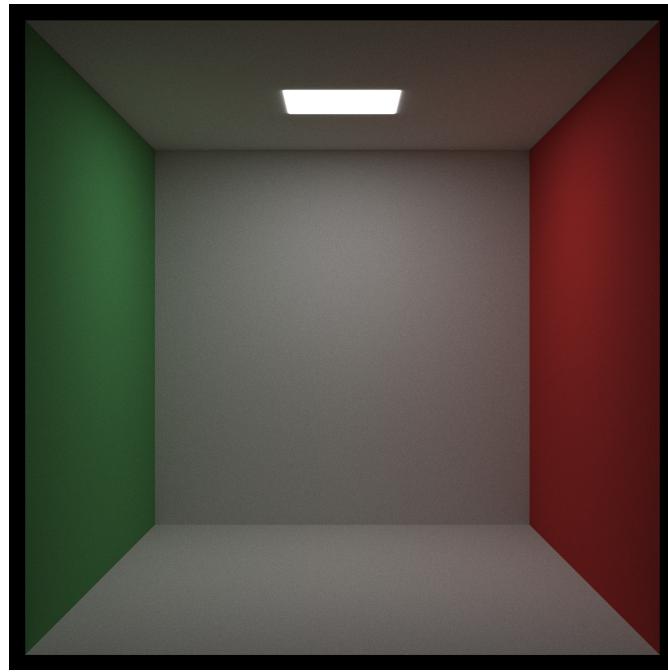


Figure 8: Cornell Box Empty

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

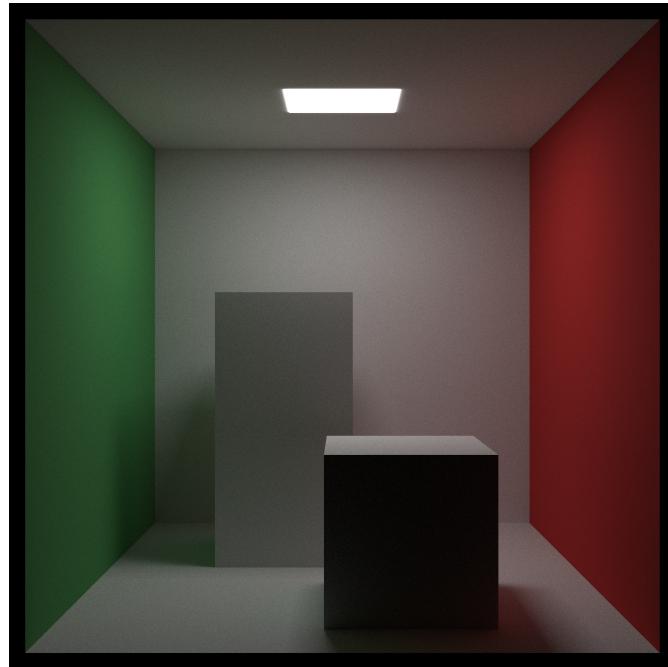


Figure 9: Cornell Box Tow Blocks

Added two boxes to the last scene.

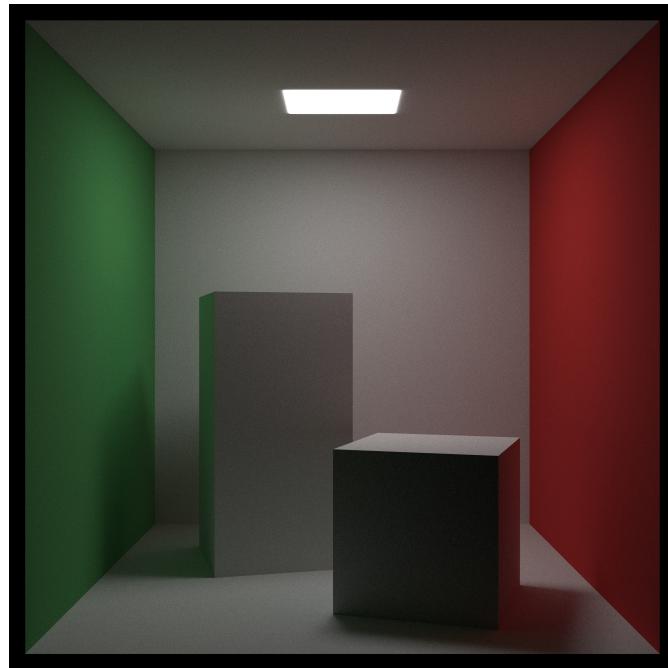


Figure 10: Cornell Box

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

2.9 Cornell Box Participating Media

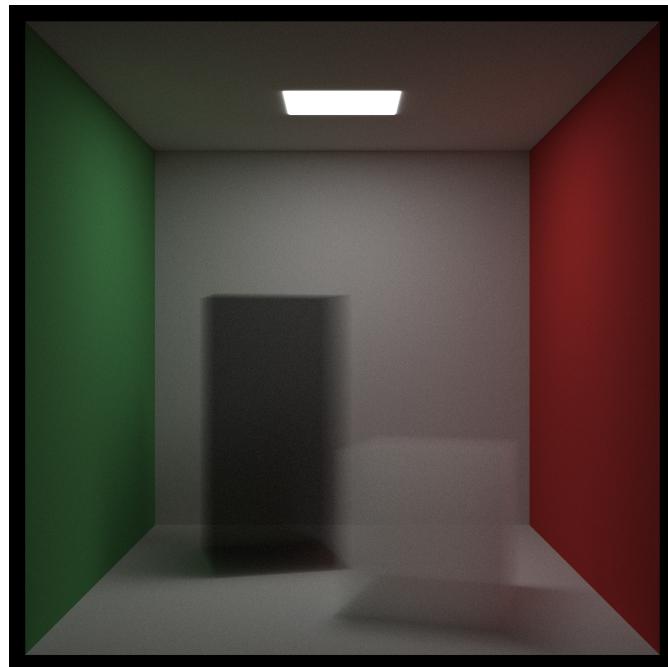


Figure 11: Cornell Box Participating Media

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

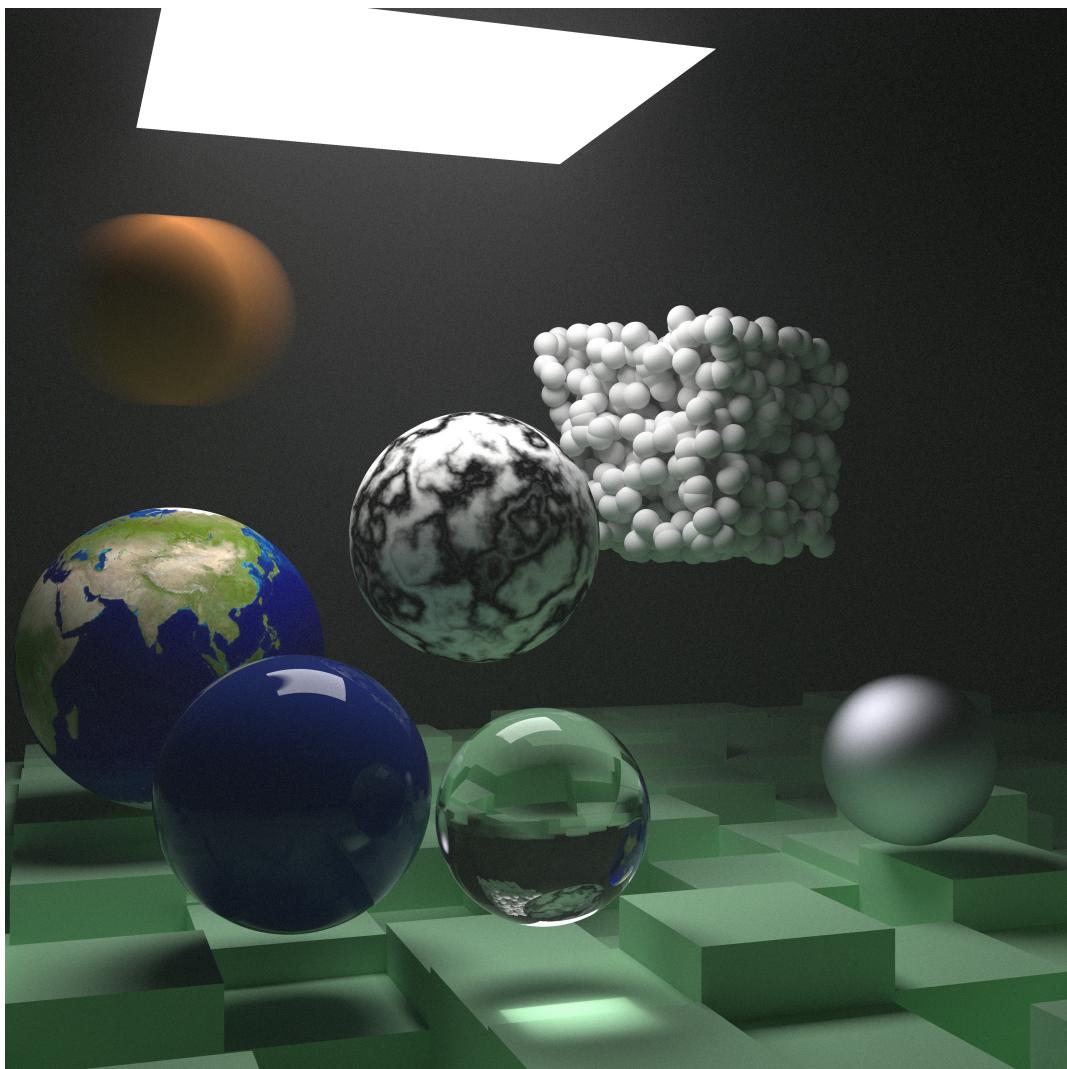


Figure 12: Book2 Final

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

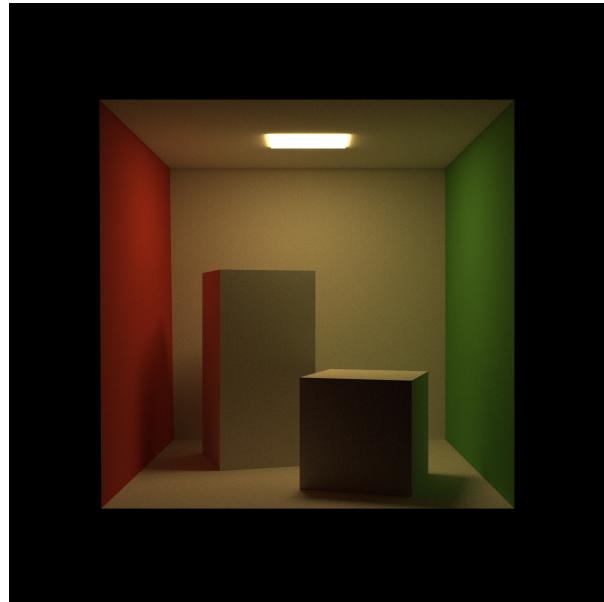


Figure 13: Test Obj

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.

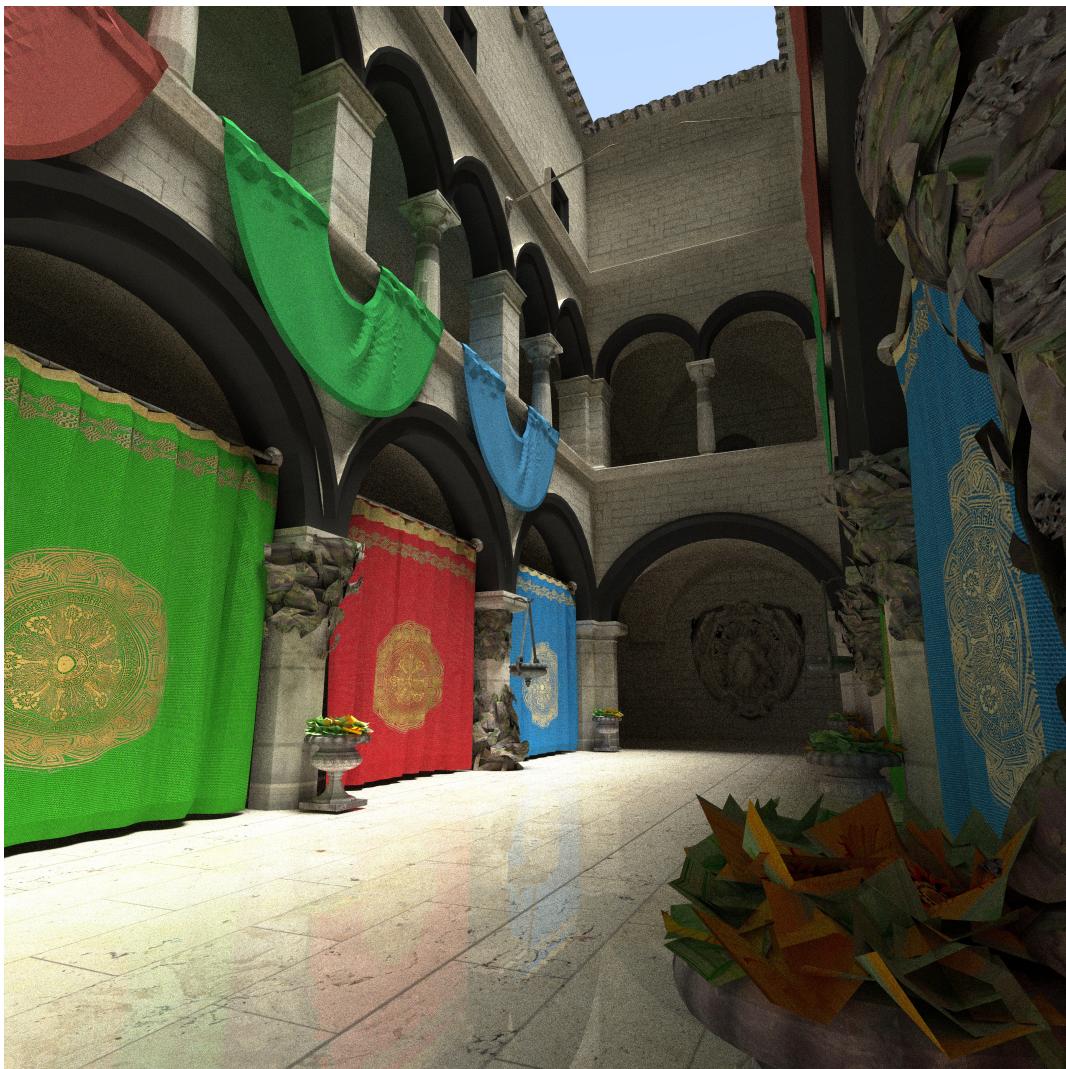


Figure 16: Sponza Crytek Sunny

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.

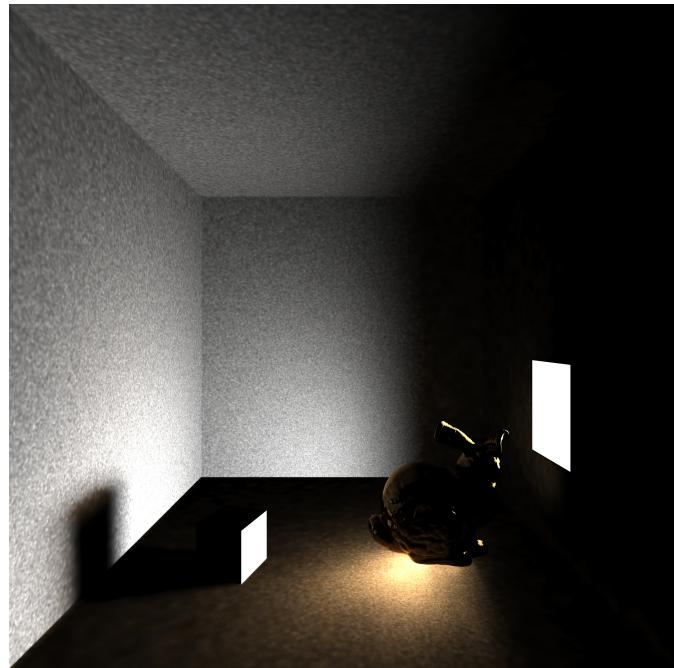


Figure 17: Photon Mapping Bunny Caustic

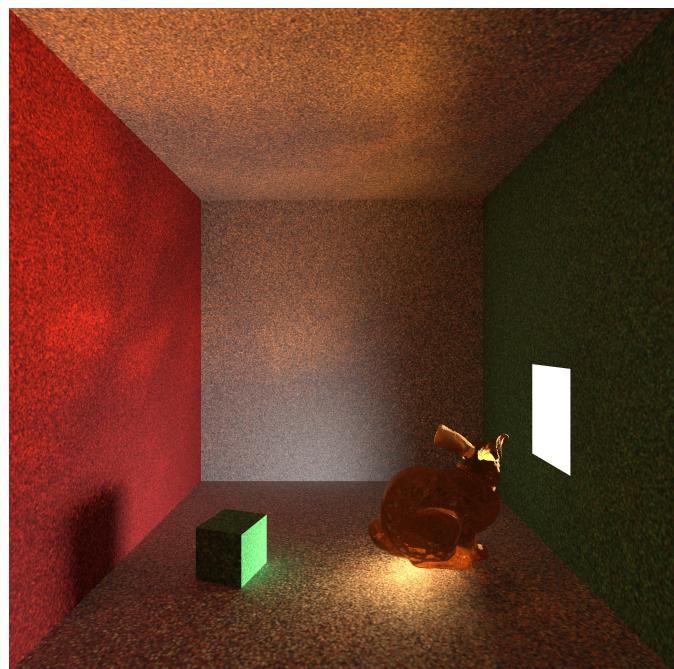


Figure 18: Photon Mapping Bunny

3 Usage

How to 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 Renderer Core:

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.

5.2.7 PhotonMap (src/core/PhotonMap)

```

1 //
2 // Created by meiyixuan on 2021-12-29.
3 //
4
5 #ifndef PROJECT_PHOTONMAP_H
6 #define PROJECT_PHOTONMAP_H
7
8 int calculate_median(int start, int end);
9
10 class PhotonMap {
11     public:
12     PhotonMap() {
13         max_photon_num = 10000;
14         photon_num = 0;
15         photon_list = new Photon[10000];
16         box_min = Vector3d(10000, 10000, 10000);
17         box_max = Vector3d(-10000, -10000, -10000);
18     };
19
20     explicit PhotonMap(int max) {
21         max_photon_num = max;
22         photon_num = 0;
23         photon_list = new Photon[max];
24         box_min = Vector3d(10000, 10000, 10000);
25         box_max = Vector3d(-10000, -10000, -10000);
26     };
27
28     ~PhotonMap() = default;
29
30     void store(Photon p) {
31         if (photon_num >= max_photon_num) return;
32         photon_list[photon_num++] = p;
33         box_min = Vector3d(fmin(box_min.x(), p.position.x()),
34                             fmin(box_min.y(), p.position.y()),
35                             fmin(box_min.z(), p.position.z()));
36         box_max = Vector3d(fmax(box_max.x(), p.position.x()),
37                             fmax(box_max.y(), p.position.y()),
38                             fmax(box_max.z(), p.position.z()));
39     }

```

```

40
41 static void median_split(Photon *temp, int start, int end, int med, int axis) {
42 // heap sort
43 int l = start, r = end;
44 while (l < r) {
45     double key = temp[r].position[axis];
46     int i = l - 1, j = r;
47     while (true) {
48         while (temp[++i].position[axis] < key);
49         while (temp[--j].position[axis] > key && j > l);
50         if (i >= j) break;
51         std::swap(temp[i], temp[j]);
52     }
53     std::swap(temp[i], temp[r]);
54     if (i >= med) r = i - 1;
55     if (i <= med) l = i + 1;
56 }
57 }

58 void balance() {
59     auto *temp = new Photon[photon_num + 1];
60     for (int i = 1; i <= photon_num; ++i) {
61         temp[i] = photon_list[i];
62     }
63     balance_segment(temp, 1, 1, photon_num);
64     delete [] temp;
65 }

66 void balance_segment(Photon *temp, int index, int start, int end) {
67     if (start == end) {
68         photon_list[index] = temp[start];
69         return;
70     }
71     int med = calculate_median(start, end);
72     int axis;
73     if (box_max.x() - box_min.x() > box_max.y() - box_min.y() &&
74     box_max.x() - box_min.x() > box_max.z() - box_min.z())
75     axis = 0;
76     else if (box_max.y() - box_min.y() > box_max.z() - box_min.z())
77     axis = 1;
78     else
79     axis = 2;
80     median_split(temp, start, end, med, axis);
81     photon_list[index] = temp[med];
82     photon_list[index].axis = axis;
83     if (start < med) {
84         double tmp = box_max[axis];
85         box_max[axis] = photon_list[index].position[axis];
86         balance_segment(temp, index * 2, start, med - 1);
87         box_max[axis] = tmp;
88     }
89     if (med < end) {
90         double tmp = box_min[axis];
91         box_min[axis] = photon_list[index].position[axis];
92         balance_segment(temp, index * 2 + 1, med + 1, end);
93         box_min[axis] = tmp;
94     }
95 }
96 }
```

```

97 }
98
99 void get_nearest_photons(NearestPhotons *nearest_photons, int index) {
100 if (index > photon_num) return;
101 Photon *photon = &photon_list[index];
102 if (index * 2 <= photon_num) {
103     double dist = nearest_photons->pos[photon->axis] - photon->position[photon->axis];
104     if (dist < 0) {
105         get_nearest_photons(nearest_photons, index * 2);
106         if (dist * dist < nearest_photons->dist_square[0])
107             get_nearest_photons(nearest_photons, index * 2 + 1);
108     } else {
109         get_nearest_photons(nearest_photons, index * 2 + 1);
110         if (dist * dist < nearest_photons->dist_square[0])
111             get_nearest_photons(nearest_photons, index * 2);
112     }
113 }
114 double dist_square = (photon->position - nearest_photons->pos).squared_length();
115 if (dist_square > nearest_photons->dist_square[0]) return;
116 if (nearest_photons->found_photons < nearest_photons->max_photons) {
117     nearest_photons->found_photons++;
118     nearest_photons->dist_square[nearest_photons->found_photons] = dist_square;
119     nearest_photons->photons[nearest_photons->found_photons] = photon;
120 } else {
121     if (!nearest_photons->heap_full) {
122         for (int i = nearest_photons->found_photons >> 1; i >= 1; --i) {
123             int par = i;
124             auto temp_photon = nearest_photons->photons[i];
125             double temp_dist_square = nearest_photons->dist_square[i];
126             while ((par << 1) <= nearest_photons->found_photons) {
127                 int j = par << 1;
128                 if (j + 1 <= nearest_photons->found_photons
129                     && nearest_photons->dist_square[j] < nearest_photons->dist_square[j + 1])
130                     j++;
131                 if (temp_dist_square >= nearest_photons->dist_square[j]) break;
132                 nearest_photons->photons[par] = nearest_photons->photons[j];
133                 nearest_photons->dist_square[par] = nearest_photons->dist_square[j];
134                 par = j;
135             }
136             nearest_photons->photons[par] = temp_photon;
137             nearest_photons->dist_square[par] = temp_dist_square;
138         }
139         nearest_photons->heap_full = true;
140     }
141     int par = 1;
142     while ((par << 1) <= nearest_photons->found_photons) {
143         int j = par << 1;
144         if (j + 1 <= nearest_photons->found_photons
145             && nearest_photons->dist_square[j] < nearest_photons->dist_square[j + 1])
146             j++;
147         if (dist_square >= nearest_photons->dist_square[j]) break;
148         nearest_photons->photons[par] = nearest_photons->photons[j];
149         nearest_photons->dist_square[par] = nearest_photons->dist_square[j];
150         par = j;
151     }
152     nearest_photons->photons[par] = photon;
153     nearest_photons->dist_square[par] = dist_square;

```

```

154     nearest_photons->dist_square[0] = nearest_photons->dist_square[1];
155 }
156 }
157
158 Color get_irradiance(Vector3d pos, Vector3d norm, double max_dist, int N) {
159     Vector3d ret(0, 0, 0);
160     NearestPhotons np;
161     np.pos = pos;
162     np.max_photons = N;
163     np.dist_square = new double[N + 1];
164     np.photons = new Photon *[N + 1];
165     np.dist_square[0] = max_dist * max_dist;
166     get_nearest_photons(&np, 1);
167     if (np.found_photons <= 0.3 * N) return ret;
168     for (int i = 1; i <= np.found_photons; ++i) {
169         auto dir = np.photons[i]->direction;
170         // filtering
171         double filter = 1.0 - (pos - np.photons[i]->position).length() / (1.1 * max_dist);
172         if (dot(norm, dir) < 0) ret = ret + np.photons[i]->power * filter;
173     }
174     ret = ret * (1.0 / (500000 * pi * np.dist_square[0]));
175     return ret;
176 }
177
178 int get_photon_num() const { return photon_num; }
179
180 private:
181     int photon_num;
182     int max_photon_num;
183     Photon *photon_list;
184     Point box_min, box_max;
185 };
186
187 int calculate_median(int start, int end) {
188     int num = end - start + 1;
189     int med;
190     int as = 1, b = 2;
191     while (as < num) {
192         as += b;
193         b *= 2;
194     }
195     if (as == num) return start + num / 2;
196     b /= 2;
197     if (as - b / 2 < num) {
198         return start + as / 2;
199     } else {
200         return start + as / 2 - (as - b / 2 - num);
201     }
202 }
203
204 #endif //PROJECT_PHOTONMAP_H

```

This is really a long file. PhotonMap is a data structure based on balanced Kd-Tree. In building stage, it is responsible for storing all photons generated by the Photon Mapper. In backward tracing stage, it serves as an oracle that returns nearest photons of any given position (as radiance). This part of the code heavily relies on Code Reference [3]'s code with some minor modifications.

5.3 External Libraries

5.3.1 stb_image (ext/stb_image/stb_image_header.h)

```
1 //  
2 // Created by meiyixuan on 2021-12-18.  
3 // This file imports stb_image.h  
4 //  
5  
6 #ifndef PROJECT_STB_IMAGE_HEADER_H  
7 #define PROJECT_STB_IMAGE_HEADER_H  
8  
9 // Disable pedantic warnings for this external library.  
10 #ifdef _MSC_VER  
11 // Microsoft Visual C++ Compiler  
12 #pragma warning (push, 0)  
13 #endif  
14  
15 #define STB_IMAGE_IMPLEMENTATION  
16  
17 #include "stb_image.h"  
18  
19 // Restore warning levels.  
20 #ifdef _MSC_VER  
21 // Microsoft Visual C++ Compiler  
22 #pragma warning (pop)  
23 #endif  
24  
25 #endif //PROJECT_STB_IMAGE_HEADER_H
```

stb_image is a nice header-only library designed for loading various kinds of images. It is used to load image into ImageTexture in the renderer. Here we only show our header, instead of the whole external library (which is thousands of lines in length).

5.3.2 tinyobjloader (ext/tinyobjloader/tiny_obj_loader_header.h)

```
1 //  
2 // Created by meiyixuan on 2021-12-20.  
3 //  
4  
5 #ifndef PROJECT_TINY_OBJ_LOADER_HEADER_H  
6 #define PROJECT_TINY_OBJ_LOADER_HEADER_H  
7  
8 #define TINYOBJLOADER_USE_DOUBLE  
9 #define TINYOBJLOADER_IMPLEMENTATION  
10 #define TINYOBJLOADER_USE_MAPBOX_EARCUT  
11  
12 #include "tiny_obj_loader.h"  
13  
14 #endif //PROJECT_TINY_OBJ_LOADER_HEADER_H
```

tinyobjloader is a header-only library for .obj import. Here, *TINYOBJLOADER_USE_DOUBLE* is defined since our renderer uses double rather than float for float point storage.

TINYOBJLOADER_USE_MAPBOX_EARCUT is defined since our mesh is triangular mesh and does not support shapes with more than three vertices. Mapbox uses Earcut algorithm to perform robust triangulation.

Note that although this algorithm is said to be robust, the real result still contains many degenerated triangles (in uv space) and needs extra preprocessing before using.

6 Renderer Core

6.1 Rendering Procedure

The rendering procedure is quite straightforward. First, in main.cpp, render workers are spawned according to platform and console arguments. Each render worker is responsible for a few lines of the final image and output its result as a .partial file. Then, packager.exe is called to assemble partial files into a .ppm image. After this step, the rendering result is finally visible. However, a typical 4K .ppm image takes around 150MB of disk space. We use convert.py to convert it into a compressed .jpg image, without losing much information. Detailed code are as follows.

6.1.1 Renderer (main.cpp) and Packager (packager.cpp)

main.cpp:

```

1 // 
2 // Created by meiyixuan on 2021-12-09.
3 // This is dev branch.
4 //
5 //***** Usage *****/
6 //
7 // Note: If compiled on Windows, flag $WINDOWS$ is automatically set. If compiled
8 //       on Linux, can use $DEBUG$ to switch between debug run (fast) or normal
9 //       run (high quality).
10 //
11 // How set scene before compile:
12 // step 1: set image settings (customization (e.g. changing aspect_ratio to
13 //           16.0 / 9.0) can be done by modifying render_scene in main.cpp)
14 // step 2: choose integrator (Photon Mapping only supports one scene, Path Tracing has step 3
15 //           of setting scene)
16 // step 3: set scene, camera and skybox (scene functions can be found in src/scenes)
17 //
18 // How to run this renderer:
19 // Windows:
20 // step 1: compile with Clion using MSVC compiler
21 // step 2: run "Project.exe 8", here 8 means using 8 threads
22 //           (after running we will get 8 .partial files)
23 // step 3: run "packager.exe 8", here 8 means combining 8 partial files
24 //           (after running we will get a .ppm image, which can be opened
25 //           using OpenSeIt)
26 // step 4: run python convert.py, which converts .ppm into .jpg
27 //           (note that opencv for python is required to run this)
28 // Linux:
29 // run "bash linux_run.sh", which compiles the renderer, uses 80 processes
30 // to run it and process result into a .ppm and a .jpg file. (Note that for
31 // default SponzaCrytek scene, about 150 GB memory is required. This requirement
32 // is proportional to the number of processes.)
33 //
34 //***** */
35 #include "Headers.h"
36
37 void render_scene(int current_id, int max_processes, const char *output_file) {

```

```

38 // Image settings
39 #if defined(WINDOWS)
40 const auto aspect_ratio = 1.0; // 16.0 / 9.0 or 1.0
41 const int image_width = 400; // 3840, 800
42 const int image_height = static_cast<int>(image_width / aspect_ratio);
43 const int samples_per_pixel = 100; // 1000, 100 (x4 if no global light)
44 const int max_depth = 10; // 100, 50
45 const int photon_map_size = 500000;
46 #elif defined(DEBUG)
47 const auto aspect_ratio = 1.0; // 16.0 / 9.0 or 1.0
48 const int image_width = 800; // 3840, 800
49 const int image_height = static_cast<int>(image_width / aspect_ratio);
50 const int samples_per_pixel = 100; // 1000, 100 (x4 if no global light)
51 const int max_depth = 10; // 100, 50
52 const int photon_map_size = 5000000;
53 #else
54 const auto aspect_ratio = 1.0; // 16.0 / 9.0 or 1.0
55 const int image_width = 3840;
56 const int image_height = static_cast<int>(image_width / aspect_ratio);
57 const int samples_per_pixel = 1000;
58 const int max_depth = 50;
59 const int photon_map_size = 20000000;
60 #endif
61
62 // image
63 auto image = std::vector<std::vector<Pixel>>();
64 for (int idx = 0; idx < image_height; ++idx) {
65     auto row = std::vector<Pixel>(image_width);
66     image.push_back(std::move(row));
67 }
68
69 // multiprocessing related (id = 0 - max_processes - 1)
70 int work_load = image_height / max_processes;
71 int start_row = image_height - 1 - work_load * current_id;
72 int end_row = current_id == max_processes - 1 ? -1 : start_row - work_load;
73
74 // Render
75 //***** Integrator *****/
76 // first specify use_photon_map() or use_path_tracing()
77 // integrator_type = use_photon_map();
78 integrator_type = use_path_tracing();
79 //*****
80 if (integrator_type == 0) {
81     // World, camera and skybox
82     // 1. Note that motion blur objects should be created with 0.0 - 1.0.
83     //     Control motion blur with camera's shutter.
84     // 2. use_global_light_skybox if no other lights enabled
85 //***** Scene *****/
86 HittableList world = sponza_crytek_scene();
87 SimpleCamera cam = sponza_crytek_camera(aspect_ratio);
88 auto skybox = sponza_crytek_skybox_cloudy();
89 //*****
90 BVHNode world_bvh(world, cam.shutter_open(), cam.shutter_close());
91
92 PathTracingIntegrator integrator(world_bvh, skybox);
93 for (int j = start_row; j > end_row; --j) {
94     std::cerr << "Scanlines remaining: " << j - end_row << '\n' << std::flush;

```

```

95     auto start = time(nullptr);
96     for (int i = 0; i < image_width; ++i) {
97         Color pixel_color(0, 0, 0);
98         for (int s = 0; s < samples_per_pixel; ++s) {
99             auto u = (i + random_double()) / (image_width - 1);
100            auto v = (j + random_double()) / (image_height - 1);
101            Ray r = cam.get_ray(u, v);
102            pixel_color += integrator.cast_ray(r, max_depth);
103        }
104        image[j][i].set(pixel_color, samples_per_pixel);
105    }
106    auto end = time(nullptr);
107    std::cerr << "loop time " << end - start << "s\n" << std::flush;
108 }
109 } else if (integrator_type == 1) {
110     // World, camera and skybox
111     HittableList world = PM_test_scene();
112     auto light = PM_test_light();
113     world.add(light);
114     SimpleCamera cam = PM_test_camera(aspect_ratio);
115     auto skybox = PM_test_skybox();
116     BVHNode world_bvh(world, cam.shutter_open(), cam.shutter_close());
117
118     // photon map and integrator
119     auto photon_map = std::make_shared<PhotonMap>(photon_map_size * 1.2);
120     auto integrator = PhotonMappingIntegrator(world, skybox, photon_map);
121
122     // generate photon map
123     Vector3d origin, direction, power = Vector3d(6, 6, 6);
124     double power_scale;
125     while (photon_map->get_photon_num() < photon_map_size) {
126         if (photon_map->get_photon_num() % 100000 == 0)
127             std::cerr << "Finished " << photon_map->get_photon_num() << " photons." << std::endl;
128         light->generate_photon(origin, direction, power_scale);
129         Ray ray(origin, direction);
130         integrator.trace_photon(ray, 10, power_scale * power);
131     }
132     while (photon_map->get_photon_num() < photon_map_size * 1.2) {
133         if (photon_map->get_photon_num() % 50000 == 0)
134             std::cerr << "Finished " << photon_map->get_photon_num() << " photons." << std::endl;
135         light->generate_photon(origin, direction, power_scale);
136         Ray ray(origin, direction);
137         integrator.trace_photon_caustic(ray, 10,
138             power_scale * power * Vector3d(0.87, 0.49, 0.173) * 12);
139     }
140     photon_map->balance();
141
142     // render
143     for (int j = start_row; j > end_row; --j) {
144         std::cerr << "Scanlines remaining: " << j - end_row << '\n' << std::flush;
145         auto start = time(nullptr);
146         for (int i = 0; i < image_width; ++i) {
147             Color pixel_color(0, 0, 0);
148             for (int s = 0; s < samples_per_pixel; ++s) {
149                 auto u = (i + random_double()) / (image_width - 1);
150                 auto v = (j + random_double()) / (image_height - 1);
151                 Ray r = cam.get_ray(u, v);

```

```

152     pixel_color += integrator.cast_ray(r, max_depth);
153 }
154 image[j][i].set(pixel_color, samples_per_pixel);
155 }
156 auto end = time(nullptr);
157 std::cerr << "loop time " << end - start << "s\n" << std::flush;
158 }
159 }
160
161 // output
162 std::ofstream file;
163 file.open(output_file, std::ios::out | std::ios::trunc);
164 if (current_id == 0)
165 file << "P3\n" << image_width << ' ' << image_height << "\n255\n";
166 for (int j = start_row; j > end_row; --j) {
167 for (int i = 0; i < image_width; ++i) {
168 image[j][i].write(file);
169 }
170 }
171 file.close();
172 std::cerr << "\nDone.\n";
173 }
174
175 int main(int argc, char *argv[]) {
176 // parse arguments
177 if (argc != 2) {
178 std::cerr << "Need to specify process count" << std::endl;
179 return 0;
180 }
181 int max_processes = parse_int(argv[1], strlen(argv[1]));
182 std::cerr << "Render will use " << max_processes << " processes." << std::endl;
183
184 // launch real renderer
185 #if defined(WINDOWS)
186
187 if (max_processes == 1) {
188 render_scene(0, 1, "1.ppm");
189 } else {
190 omp_set_num_threads(max_processes);
191 #pragma omp parallel for
192 for (int thread_idx = 0; thread_idx < max_processes; ++thread_idx) {
193 std::string file_name = std::to_string(thread_idx) + ".partial";
194 render_scene(thread_idx, max_processes, file_name.c_str());
195 }
196 }
197
198 #else
199
200 if (max_processes == 1) {
201 render_scene(0, 1, "1.ppm");
202 } else {
203 std::vector<pid_t> pid_list;
204 for (int process_idx = 0; process_idx < max_processes; ++process_idx) {
205 std::string file_name = std::to_string(process_idx) + ".partial";
206 pid_t pid = fork();
207 if (pid != 0) {
208 // parent

```

```

209     pid_list.push_back(pid);
210     continue;
211 } else {
212     // child
213     std::cerr << "Worker " << process_idx << " initialized on " << pid << std::endl;
214     render_scene(process_idx, max_processes, file_name.c_str());
215     return 0;
216 }
217 }
218 for (auto pid: pid_list) {
219     waitpid(pid, NULL, 0);
220 }
221 }
222
223 #endif
224 }
```

packager.cpp:

```

1 //
2 // Created by meiyixuan on 2021-12-14.
3 //
4
5 #include "Headers.h"
6
7 using namespace std;
8
9 int main(int argc, char *argv[]) {
10    // check argument count
11    if (argc != 2) {
12        cout << "Need to specify the number of partial files" << endl;
13        return 0;
14    }
15    int file_count = parse_int(argv[1], strlen(argv[1]));
16
17    // open output
18    ofstream output;
19    output.open("1.ppm", ios::out | ios::trunc);
20
21    // parse partial files
22    for (int idx = 0; idx < file_count; ++idx) {
23        // open input
24        ifstream input;
25        string input_file_name = to_string(idx) + ".partial";
26        input.open(input_file_name.c_str());
27
28        // parse
29        string line;
30        while (getline(input, line)) {
31            output << line << "\n";
32        }
33
34        // clean up
35        input.close();
36    }
37    output.close();
38 }
```

6.1.2 Other files

CMakeList.txt

```
1 cmake_minimum_required(VERSION 3.19)
2 project(Project)
3
4 set(CMAKE_CXX_STANDARD 14)
5
6 IF (WIN32)
7 add_definitions(-D WINDOWS)
8 set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} /openmp")
9 ELSE()
10 add_definitions(-D DEBUG)
11 set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++17 -O2 -pthread -fopenmp")
12 ENDIF()
13
14 add_executable(Project main.cpp)
15 add_executable(packager packager.cpp)
```

linux_run.sh

```
1 cmake .
2 make clean
3 make
4 ./Project 80
5 rm 1.ppm
6 rm 1.jpg
7 ./packager 80
8 rm *.partial
9 python convert.py
```

converter.py

```
1 import cv2
2
3 i = cv2.imread('1.ppm')
4 cv2.imwrite('1.jpg', i)
```

6.2 Integrator

6.2.1 Base Integrator (src/core/Integrator.h)

```
1 /**
2 // Created by meiyixuan on 2021-12-15.
3 /**
4
5 #ifndef PROJECT_INTEGRATOR_H
6 #define PROJECT_INTEGRATOR_H
7
8 class Integrator {
9 public:
10     Integrator() = delete;
11
12     explicit Integrator(const Accelerator &scene, const Skybox &_skybox)
13     : world(scene), skybox(_skybox) {}
14
15     virtual Color cast_ray(const Ray &r, int remaining_bounce) const = 0;
16 }
```

```

17     protected:
18     const Accelerator &world;
19     const Skybox &skybox;
20 };
21
22 #endif //PROJECT_INTEGRATOR_H

```

This is the base class for all integrators. A integrator integrate light intensity along given light path and returns raw pixel color. *world* represents the whole scene (usually a BVH node) and *skybox* is a global (usually uniform) light source. Typical integrators include Path Tracing, Photon Mapping, Progressive Photon Mapping, etc. In this project we only implement the first two.

6.2.2 Path Tracing Integrator (src/integrator/PathTracingIntegrator.h)

```

1 //
2 // Created by meiyixuan on 2021-12-18.
3 //
4
5 #ifndef PROJECT_PATHTRACINGINTEGRATOR_H
6 #define PROJECT_PATHTRACINGINTEGRATOR_H
7
8 class PathTracingIntegrator : Integrator {
9 public:
10 PathTracingIntegrator() = delete;
11
12 explicit PathTracingIntegrator(const Accelerator &scene, const Skybox &skybox)
13 : Integrator(scene, skybox) {}
14
15 Color cast_ray(const Ray &r, int remaining_bounce) const override {
16     Hit hit;
17
18     // If we've exceeded the ray bounce limit, no more light is gathered.
19     if (remaining_bounce <= 0) return {0, 0, 0};
20     hit.remaining_bounce = remaining_bounce;
21
22     // If hit nothing, return background
23     if (!world.hit(r, TMIN, inf, hit)) return skybox.get_color(r);
24
25     // hit something
26     Ray scattered_ray;
27     Color emit_color = hit.mat_ptr->emit(hit.u, hit.v, hit.hit_point);
28
29     // no scattering
30     if (!hit.mat_ptr->scatter(r, hit, scattered_ray))
31         return emit_color;
32
33     // scattering
34     Color attenuation = hit.mat_ptr->brdf(r, scattered_ray, hit);
35     Color scatter_color = cast_ray(scattered_ray, remaining_bounce - 1);
36     return emit_color + attenuation * scatter_color;
37 }
38 };
39
40 #endif //PROJECT_PATHTRACINGINTEGRATOR_H

```

A Path Tracing integrator reflects and refracts just like normal Ray Tracing integrator. However, when a ray hits a diffuse surface (e.g. Lambertian), instead of sending test rays enumerating lights, path tracing samples a random

reflection direction with cos probability and trace the resulting ray. In our implementation, sampling is done in Material. Therefore, Path Tracing Integrator is very short and neat.

6.2.3 Photon Mapping Integrator (src/integrator/PhotonMappingIntegrator.h)

```

1 // 
2 // Created by meiyixuan on 2021-12-29.
3 //
4
5 #ifndef PROJECT_PHOTONMAPPINGINTEGRATOR_H
6 #define PROJECT_PHOTONMAPPINGINTEGRATOR_H
7
8 class PhotonMappingIntegrator : public Integrator {
9     public:
10     PhotonMappingIntegrator(const Accelerator &scene, const Skybox &skybox, std::shared_ptr<PhotonMap>
11         _map)
12     : Integrator(scene, skybox), photon_map(std::move(_map)) {}
13
14     Color cast_ray(const Ray &r, int remaining_bounce) const override {
15         Hit hit;
16         if (world.hit(r, TMIN, inf, hit)) {
17             Ray scattered_ray;
18             Color emit_color = hit.mat_ptr->emit(hit.u, hit.v, hit.hit_point);
19             if (remaining_bounce >= 0 && hit.mat_ptr->scatter(r, hit, scattered_ray)) {
20                 auto attenuation = hit.mat_ptr->brdf(r, scattered_ray, hit);
21                 Color light_color;
22                 if (hit.scatter_mode == 0) {
23                     light_color = photon_map->get_irradiance(hit.hit_point, hit.normal, 0.1, 50);
24                 } else {
25                     light_color = cast_ray(scattered_ray, remaining_bounce - 1);
26                 }
27                 return emit_color + light_color * attenuation;
28             } else {
29                 return emit_color;
30             }
31         } else {
32             return skybox.get_color(r);
33         }
34     }
35
36     void trace_photon(const Ray &ray, int remaining_bounce, Vector3d power) {
37         Hit hit;
38         if (world.hit(ray, TMIN, inf, hit)) {
39             Ray scattered_ray;
40             if (remaining_bounce > 0 && hit.mat_ptr->scatter(ray, hit, scattered_ray)) {
41                 // store photon at diffuse surface
42                 if (hit.scatter_mode == 0) {
43                     photon_map->store(Photon(hit.hit_point, ray.direction(), power));
44                 }
45                 // generate a new photon
46                 Color attenuation = hit.mat_ptr->brdf(ray, scattered_ray, hit);
47                 trace_photon(scattered_ray, remaining_bounce - 1, power * attenuation);
48             }
49         }
50     }
51     void trace_photon_caustic(const Ray &ray, int remaining_bounce, Vector3d power) {
```

```

52     Hit hit;
53     if (world.hit(ray, TMIN, inf, hit)) {
54         Ray scattered_ray;
55         if (remaining_bounce > 0 && hit.mat_ptr->scatter(ray, hit, scattered_ray)) {
56             if (hit.scatter_mode == 1 || hit.scatter_mode == 2) {
57                 trace_photon(scattered_ray, remaining_bounce - 1, power);
58             } else {
59                 if (remaining_bounce == 10) return;
60                 photon_map->store(Photon(hit.hit_point, ray.direction(), power));
61             }
62         }
63     }
64 }
65
66 private:
67 std::shared_ptr<PhotonMap> photon_map;
68 };
69
70 #endif //PROJECT_PHOTONMAPPINGINTEGRATOR_H

```

Photon Mapping is also known as Photon Mapping Path Tracing (PMPT). Different from normal Path Tracing integrators, instead of sampling a random direction at every diffuse surface, it queries a photon map for light intensity information, which gives faster convergence when light is very small. PM has two passes. The first pass is forward path tracing. It sends out rays (i.e. photons) from every light in the scene, let them reflect and refract and store their intensity and position whenever they hit a diffuse surface. Forward ray tracing builds up a photon map (a Kd-tree of \langle position, intensity \rangle tuples). The second pass is backward path tracing, this pass is identical to normal Path Tracing except that it queries the photon map and uses nearest photons' intensity as light intensity of diffuse hit-point. Note that we need to set a max radius for nearest photon query to prevent points in shadows being lit by faraway photons.

6.3 Accelerator

6.3.1 Base Accelerator (src/core/Accelerator.h)

```

1 // 
2 // Created by meiyixuan on 2021-12-10.
3 // This file contains all accelerating data structures.
4 //
5
6 #ifndef PROJECT_ACCELERATOR_H
7 #define PROJECT_ACCELERATOR_H
8
9 using std::shared_ptr;
10 using std::make_shared;
11
12 class Accelerator : public Hittable {
13     // Base class for all accelerating data structures
14 };
15
16 #endif //PROJECT_ACCELERATOR_H

```

Accelerator, for compatibility, is derived from Hittable.

6.3.2 Hittable List (src/accelerator/HittableList.h)

```

1 // 
2 // Created by meiyixuan on 2021-12-18.
3 //
4
5 #ifndef PROJECT_HITTABLELIST_H
6 #define PROJECT_HITTABLELIST_H
7
8 class HittableList : public Accelerator {
9     public:
10     HittableList() = default;
11
12     void clear() { hittable_list.clear(); }
13
14     void add(const shared_ptr<Hittable> &obj) { hittable_list.push_back(obj); }
15
16     bool hit(const Ray &r, double t_min, double t_max, Hit &hit) const override {
17         // initialize
18         Hit obj_hit;
19         bool has_hit = false;
20         auto closest_t = t_max;
21
22         // check intersect status
23         for (const auto &hittable: hittable_list) {
24             if (hittable->hit(r, t_min, closest_t, obj_hit)) {
25                 has_hit = true;
26                 closest_t = obj_hit.t;
27                 hit = obj_hit;
28             }
29         }
30
31         return has_hit;
32     }
33
34     bool bounding_box(double time0, double time1, AABB &output_box) const override {
35         // check if there are objects
36         if (hittable_list.empty()) return false;
37
38         AABB temp_box;
39         bool first_box_flag = true;
40
41         for (const auto &object: hittable_list) {
42             // if there are infinitely large object, return false
43             if (!object->bounding_box(time0, time1, temp_box)) return false;
44
45             // check if first box
46             output_box = first_box_flag ? temp_box : surrounding_box(output_box, temp_box);
47             first_box_flag = false;
48         }
49         return true;
50     }
51
52     protected:
53     friend class BVHNode;
54
55     std::vector<std::shared_ptr<Hittable>> hittable_list;
56 };
57

```

```
58 #endif //PROJECT_HITTABLELIST_H
```

The simplest form of integrator is a list. Although it has O(n) query complexity, it is still very efficient when objects are not too many and can be used to initialize other accelerators (like BVH).

6.3.3 BVH (src/accelerator/BVH.h)

```
1 //  
2 // Created by meiyixuan on 2021-12-18.  
3 //  
4  
5 #ifndef PROJECT_BVH_H  
6 #define PROJECT_BVH_H  
7  
8 class BVHNode : public Accelerator {  
9     public:  
10         BVHNode() = default;  
11  
12         BVHNode(HittableList &list, double time0, double time1)  
13             : BVHNode(list.hittable_list, 0, list.hittable_list.size(), time0, time1) {}  
14  
15         BVHNode(std::vector<shared_ptr<Hittable>> &src_objects, size_t start, size_t end,  
16                  double time0, double time1) {  
17             // surface-area-heuristic BVH build  
18             // start inclusive, end exclusive  
19  
20             // special case  
21             if (src_objects.size() == 1) {  
22                 left = src_objects[0];  
23                 right = nullptr;  
24                 src_objects[0]->bounding_box(time0, time1, box);  
25                 return;  
26             }  
27  
28             // initialize  
29             size_t n = end - start;  
30             std::vector<AABB> boxes = std::vector<AABB>(n);  
31             std::vector<double> left_area = std::vector<double>(n);  
32             std::vector<double> right_area = std::vector<double>(n);  
33             AABB main_box;  
34             src_objects[start]->bounding_box(time0, time1, main_box);  
35             for (int i = 1; i < n; ++i) {  
36                 AABB new_box;  
37                 src_objects[start + i]->bounding_box(time0, time1, new_box);  
38                 main_box = surrounding_box(new_box, main_box);  
39             }  
40  
41             // choose split axis  
42             int axis = main_box.longest_axis();  
43             if (axis == 0)  
44                 sort(src_objects.begin() + static_cast<long long>(start),  
45                       src_objects.begin() + static_cast<long long>(end),  
46                       box_x_compare);  
47             else if (axis == 1)  
48                 sort(src_objects.begin() + static_cast<long long>(start),  
49                       src_objects.begin() + static_cast<long long>(end),  
50                       box_y_compare);
```

```

51
52     sort(src_objects.begin() + static_cast<long long>(start) ,
53     src_objects.begin() + static_cast<long long>(end) ,
54     box_z_compare);
55     for (int i = 0; i < n; ++i)
56     src_objects[start + i]→bounding_box(time0, time1, boxes[i]);
57
58 // left area size
59 left_area[0] = boxes[0].area();
60 AABB left_box = boxes[0];
61 for (size_t i = 1; i < n - 1; ++i) {
62     left_box = surrounding_box(left_box, boxes[i]);
63     left_area[i] = left_box.area();
64 }
65
66 // right area size
67 right_area[n - 1] = boxes[n - 1].area();
68 AABB right_box = boxes[n - 1];
69 for (size_t i = n - 2; i > 0; --i) {
70     right_box = surrounding_box(right_box, boxes[i]);
71     right_area[i] = right_box.area();
72 }
73
74 // use SAH heuristic to find split
75 double min_SAH = inf;
76 size_t min_SAH_idx;
77 for (size_t i = 0; i < n - 1; ++i) {
78     double SAH = static_cast<double>(i) * left_area[i] +
79     static_cast<double>(n - i - 1) * right_area[i + 1];
80     if (SAH < min_SAH) {
81         min_SAH_idx = i;
82         min_SAH = SAH;
83     }
84 }
85
86 // split
87 if (min_SAH_idx == 0)
88     left = src_objects[start];
89 else
90     left = std::make_shared<BVHNode>(src_objects, start, start + min_SAH_idx + 1,
91     time0, time1);
92     if (min_SAH_idx == n - 2)
93     right = src_objects[start + min_SAH_idx + 1];
94     else
95     right = std::make_shared<BVHNode>(src_objects, start + min_SAH_idx + 1, end,
96     time0, time1);
97     box = main_box;
98 }
99
100 bool hit(const Ray &ray, double t_min, double t_max, Hit &hit) const override {
101     // if input ray doesn't hit current node
102     if (!box.hit(ray, t_min, t_max))
103     return false;
104
105     // recursively check two child nodes
106     bool hit_left = false, hit_right = false;
107     if (left) hit_left = left→hit(ray, t_min, t_max, hit);

```

```

108     if (right) hit_right = right->hit(ray, t_min, hit_left ? hit.t : t_max, hit);
109     return hit_left || hit_right;
110 }
111
112 bool bounding_box(double time0, double time1, AABB &output_box) const override {
113     output_box = box;
114     return true;
115 }
116
117 protected:
118 // can be either accelerator or object
119 shared_ptr<Hittable> left;
120 shared_ptr<Hittable> right;
121 AABB box;
122
123 private:
124     static inline bool box_compare(const shared_ptr<Hittable> &a,
125                                   const shared_ptr<Hittable> &b, int axis) {
126         AABB box_a;
127         AABB box_b;
128
129         if (!a->bounding_box(0, 0, box_a) || !b->bounding_box(0, 0, box_b))
130             std::cerr << "No bounding box found.\n";
131
132         return box_a.min().val[axis] < box_b.min().val[axis];
133     }
134
135     static inline bool box_x_compare(const shared_ptr<Hittable> &a, const shared_ptr<Hittable> &b) {
136         return box_compare(a, b, 0);
137     }
138
139     static inline bool box_y_compare(const shared_ptr<Hittable> &a, const shared_ptr<Hittable> &b) {
140         return box_compare(a, b, 1);
141     }
142
143     static inline bool box_z_compare(const shared_ptr<Hittable> &a, const shared_ptr<Hittable> &b) {
144         return box_compare(a, b, 2);
145     }
146
147 };
148
149 #endif //PROJECT_BVH_H

```

This file contains a BVH accelerator. It can be initialized using HittableList. Unlike Kd-tree, there is no guarantee for balanced implementation. Therefore, we use surface-area-heuristic (SAH, line 15 - 98) that empirically generates a spatially balanced tree.

6.3.4 Kd-Tree

In general, Kd-Tree can also be used in accelerating hit calculation. However, since we have a very fast BVH based on AABB and SAH, we no longer need Kd-Tree for this purpose. Instead, Kd-Tree is used in Photon Map for fast nearest neighbor calculation. For details see 5.2.7.

6.4 Hardware Acceleration

main.cpp

```
1 // launch real renderer
2 #if defined(WINDOWS)
3
4 if (max_processes == 1) {
5     render_scene(0, 1, "1.ppm");
6 } else {
7     omp_set_num_threads(max_processes);
8     #pragma omp parallel for
9     for (int thread_idx = 0; thread_idx < max_processes; ++thread_idx) {
10         std::string file_name = std::to_string(thread_idx) + ".partial";
11         render_scene(thread_idx, max_processes, file_name.c_str());
12     }
13 }
14
15 #else
16
17 if (max_processes == 1) {
18     render_scene(0, 1, "1.ppm");
19 } else {
20     std::vector<pid_t> pid_list;
21     for (int process_idx = 0; process_idx < max_processes; ++process_idx) {
22         std::string file_name = std::to_string(process_idx) + ".partial";
23         pid_t pid = fork();
24         if (pid != 0) {
25             // parent
26             pid_list.push_back(pid);
27             continue;
28         } else {
29             // child
30             std::cerr << "Worker " << process_idx << " initialized on " << pid << std::endl;
31             render_scene(process_idx, max_processes, file_name.c_str());
32             return 0;
33         }
34     }
35     for (auto pid: pid_list) {
36         waitpid(pid, NULL, 0);
37     }
38 }
39
40#endif
```

For ray tracing, the most natural way of hardware acceleration is GPU acceleration, especially by using RT cores in NVIDIA GPUs. However, GPU acceleration requires coding with CUDA and managing VRAM, which is more complicated and harder to debug. Therefore, in our implementation we use CPU multi-processing (or multi-threading). This acceleration method is quite straightforward and requires only minor changes to the code (tens of lines).

For Windows platforms, we use OpenMP to spawn multiple render worker threads. By adding only one line, we can achieve almost linear acceleration. However, for Linux based OS, things are a little bit trickier. C++ standard libraries have locks within its implementation (global, thread-wise lock). It will block access from other threads even if all accesses are read accesses. The result is that all cores are busy waiting and no acceleration is achieved. Therefore, for Linux, we use multiprocessing. Render worker processes are forked from main process and gathered

after exit. In this way, we can still achieve linear acceleration. For all experiments, we run the renderer with 80 workers on a server with 128 cores and 512GB memory. Hardware acceleration makes things faster about 80x.

7 Objects

In this section, we introduce objects defined in our renderer. This ranges from simple shapes to complex model, as well as transforms.

7.1 Hittable Objects

7.1.1 Base Hittable (src/core/Hittable.h)

```
1 //  
2 // Created by meiyixuan on 2021-12-10.  
3 // This file contains all hittable objects.  
4 //  
5  
6 #ifndef PROJECT_HITTABLE_H  
7 #define PROJECT_HITTABLE_H  
8  
9 // base class for all hittable objects  
10 class Hittable {  
11     public:  
12     Hittable() : mat_ptr(nullptr) {}  
13  
14     explicit Hittable(std::shared_ptr<Material> m) : mat_ptr(std::move(m)) {}  
15  
16     virtual bool hit(const Ray &ray, double t_min, double t_max, Hit &hit) const = 0;  
17  
18     virtual bool bounding_box(double time0, double time1, AABB &output_box) const = 0;  
19  
20     protected:  
21     shared_ptr<Material> mat_ptr;  
22 };  
23  
24 #endif //PROJECT_HITTABLE_H
```

All Hittables have two virtual functions to be defined. The first one is *hit*, which determines whether a given ray intersects with the Hittable or not. The second one is *boundingbox*, which returns a proper AABB that surrounds the Hittable. The first one is used by Integrators and the second one by BVH.

7.1.2 Sphere (src/hittable/Sphere.h)

```
1 //  
2 // Created by meiyixuan on 2021-12-18.  
3 //  
4  
5 #ifndef PROJECT_SPHERE_H  
6 #define PROJECT_SPHERE_H  
7  
8 class Sphere : public Hittable {  
9     public:  
10         // constructors  
11         Sphere() : r{0} {}
```

```

12
13     Sphere(Point center, double radius, std::shared_ptr<Material> m) : Hittable(std::move(m)), 
14     c{center}, r{radius} {};
15
16     // utilities
17     Point center() const { return c; }
18
19     double radius() const { return r; }
20
21     bool hit(const Ray &ray, double t_min, double t_max, Hit &hit) const override {
22         // calculate
23         Vector3d oc = ray.origin() - c;
24         auto _a = ray.direction().squared_length();
25         auto _b = dot(oc, ray.direction());
26         auto _c = oc.squared_length() - r * r;
27         auto discriminant = _b * _b - _a * _c;
28         if (discriminant < 0) return false;
29         auto sqrt_d = sqrt(discriminant);
30
31         // fill intersect
32         auto root = (-_b - sqrt_d) / _a;
33         if (root > t_min && root < t_max) {
34             // small root is fine
35             hit.t = root;
36             hit.hit_point = ray.at(root);
37             Vector3d normal = (hit.hit_point - c) / r;
38             hit.set_face_normal(ray, normal);
39             hit.mat_ptr = mat_ptr;
40             get_sphere_uv(normal, hit.u, hit.v);
41             return true;
42         } else {
43             // may use larger root
44             root = (-_b + sqrt_d) / _a;
45             if (root > t_min && root < t_max) {
46                 hit.t = root;
47                 hit.hit_point = ray.at(root);
48                 Vector3d normal = (hit.hit_point - c) / r;
49                 hit.set_face_normal(ray, normal);
50                 hit.mat_ptr = mat_ptr;
51                 get_sphere_uv(normal, hit.u, hit.v);
52                 return true;
53             } else { return false; }
54         }
55     }
56
57     bool bounding_box(double time0, double time1, AABB &output_box) const override {
58         output_box = AABB(c - Vector3d(r, r, r), c + Vector3d(r, r, r));
59         return true;
60     }
61
62     private:
63     Point c;
64     double r;
65
66     static void get_sphere_uv(const Point &outward_normal, double &u, double &v) {
67         // outward_normal: a given point on the sphere of radius one, centered at the origin.
68         // u: returned value [0,1] of angle around the Y axis from X=-1.

```

```

69 // v: returned value [0,1] of angle from Y=-1 to Y=+1.
70
71 // calculate parameter
72 auto theta = acos(-outward_normal.y());
73 auto phi = atan2(-outward_normal.z(), outward_normal.x()) + pi;
74
75 // calculate uv
76 u = phi / (2 * pi);
77 v = theta / pi;
78 }
79 };
80
81 #endif //PROJECT_SPHERE_H

```

The simplest object is a sphere. For *hit*, we use code form Real Time Rendering and Ray Tracing Mini-books.

7.1.3 Moving Sphere (src/hittable/MovingSphere.h)

```

1 //
2 // Created by meiyixuan on 2021-12-18.
3 //
4
5 #ifndef PROJECT_MOVINGSHERE_H
6 #define PROJECT_MOVINGSHERE_H
7
8 class MovingSphere : public Hittable {
9 public:
10 MovingSphere() = default;
11
12 MovingSphere(Point cen0, Point cen1, double _time0, double _time1, double r, const shared_ptr<Material> &m)
13 : Hittable(m), center0(cen0), center1(cen1), time0(_time0), time1(_time1), radius(r) {};
14
15 bool hit(const Ray &ray, double t_min, double t_max, Hit &hit) const override {
16     // calculate
17     Vector3d oc = ray.origin() - center(ray.time());
18     auto _a = ray.direction().squared_length();
19     auto _b = dot(oc, ray.direction());
20     auto _c = oc.squared_length() - radius * radius;
21     auto discriminant = _b * _b - _a * _c;
22     if (discriminant < 0) return false;
23     auto sqrt_d = sqrt(discriminant);
24
25     // fill intersect
26     auto root = (-_b - sqrt_d) / _a;
27     if (root > t_min && root < t_max) {
28         // small root is fine
29         hit.t = root;
30         hit.hit_point = ray.at(root);
31         Vector3d normal = (hit.hit_point - center(ray.time())) / radius;
32         hit.set_face_normal(ray, normal);
33         hit.mat_ptr = mat_ptr;
34         return true;
35     } else {
36         // may use larger root
37         root = (-_b + sqrt_d) / _a;
38         if (root > t_min && root < t_max) {

```

```

39         hit.t = root;
40         hit.hit_point = ray.at(root);
41         Vector3d normal = (hit.hit_point - center(ray.time())) / radius;
42         hit.set_face_normal(ray, normal);
43         hit.mat_ptr = mat_ptr;
44         return true;
45     } else { return false; }
46 }
47 }

48 bool bounding_box(double _time0, double _time1, AABB &output_box) const override {
49     AABB box0(center(_time0) - Vector3d(radius, radius, radius),
50               center(_time0) + Vector3d(radius, radius, radius));
51     AABB box1(center(_time1) - Vector3d(radius, radius, radius),
52               center(_time1) + Vector3d(radius, radius, radius));
53     output_box = surrounding_box(box0, box1);
54     return true;
55 }
56 }

57 Point center(double time) const {
58     return center0 + ((time - time0) / (time1 - time0)) * (center1 - center0);
59 }
60 }

61 public:
62     Point center0, center1;
63     double time0{}, time1{};
64     double radius{};
65 };
66 }

67 #endif //PROJECT_MOVINGSPHERE_H

```

Moving spheres have center position as a function of time. Since Rays have time within it, we can use the time to compute the center and reduce it to the problem of checking intersection between static spheres and rays.

7.1.4 Rectangles (src/hittable/Rectangle.h)

```

1 // 
2 // Created by meiyixuan on 2021-12-18.
3 //
4
5 #ifndef PROJECT_RECTANGLE_H
6 #define PROJECT_RECTANGLE_H
7
8 class XYRectangle : public Hittable {
9     public:
10     XYRectangle() = default;
11
12     XYRectangle(double _x0, double _x1, double _y0, double _y1, double _z, std::shared_ptr<Material>
13                 mat_ptr)
14     : Hittable(std::move(mat_ptr)), x0(_x0), x1(_x1), y0(_y0), y1(_y1), z(_z) {}
15
16     bool hit(const Ray &ray, double t_min, double t_max, Hit &hit) const override {
17         // check if hit plane
18         auto t = (z - ray.origin().z()) / ray.direction().z();
19         if (t < t_min || t > t_max) return false;
20
21         // check if hit rect

```

```

21 auto x = ray.origin().x() + t * ray.direction().x();
22 auto y = ray.origin().y() + t * ray.direction().y();
23 if (x < x0 || x > x1 || y < y0 || y > y1) return false;
24
25 // hit
26 hit.u = (x - x0) / (x1 - x0);
27 hit.v = (y - y0) / (y1 - y0);
28 hit.t = t;
29 auto outward_normal = Vector3d(0, 0, 1);
30 hit.set_face_normal(ray, outward_normal);
31 hit.mat_ptr = mat_ptr;
32 hit.hit_point = ray.at(t);
33 return true;
34 }
35
36 bool bounding_box(double time0, double time1, AABB &output_box) const override {
37     output_box = AABB(Point(x0, y0, z - 0.0001), Point(x1, y1, z + 0.0001));
38     return true;
39 }
40
41 private:
42 double x0{}, x1{}, y0{}, y1{};
43 double z{};
44 };
45
46 class XZRectangle : public Hittable {
47 public:
48 XZRectangle() = default;
49
50 XZRectangle(double _x0, double _x1, double _z0, double _z1, double _y, shared_ptr<Material> mat_ptr)
51 : Hittable(std::move(mat_ptr)), x0(_x0), x1(_x1), z0(_z0), z1(_z1), y(_y) {};
52
53 bool hit(const Ray &ray, double t_min, double t_max, Hit &hit) const override {
54     // check if hit plane
55     auto t = (y - ray.origin().y()) / ray.direction().y();
56     if (t < t_min || t > t_max) return false;
57
58     // check if hit rectangle
59     auto x = ray.origin().x() + t * ray.direction().x();
60     auto z = ray.origin().z() + t * ray.direction().z();
61     if (x < x0 || x > x1 || z < z0 || z > z1) return false;
62
63     // hit
64     hit.u = (x - x0) / (x1 - x0);
65     hit.v = (z - z0) / (z1 - z0);
66     hit.t = t;
67     auto outward_normal = Vector3d(0, 1, 0);
68     hit.set_face_normal(ray, outward_normal);
69     hit.mat_ptr = mat_ptr;
70     hit.hit_point = ray.at(t);
71     return true;
72 }
73
74 bool bounding_box(double time0, double time1, AABB &output_box) const override {
75     output_box = AABB(Point(x0, y - 0.0001, z0), Point(x1, y + 0.0001, z1));
76     return true;
77 }
```

```

78
79     private:
80     double x0{}, x1{}, z0{}, z1{};
81     double y{};
82 };
83
84 class YZRectangle : public Hittable {
85     public:
86     YZRectangle() = default;
87
88     YZRectangle(double _y0, double _y1, double _z0, double _z1, double _x, shared_ptr<Material> mat_ptr)
89     : Hittable(std::move(mat_ptr)), y0(_y0), y1(_y1), z0(_z0), z1(_z1), x(_x) {};
90
91     bool hit(const Ray &ray, double t_min, double t_max, Hit &hit) const override {
92         // check if hit plane
93         auto t = (x - ray.origin().x()) / ray.direction().x();
94         if (t < t_min || t > t_max) return false;
95
96         // check if hit rectangle
97         auto y = ray.origin().y() + t * ray.direction().y();
98         auto z = ray.origin().z() + t * ray.direction().z();
99         if (y < y0 || y > y1 || z < z0 || z > z1) return false;
100
101        // hit
102        hit.u = (y - y0) / (y1 - y0);
103        hit.v = (z - z0) / (z1 - z0);
104        hit.t = t;
105        auto outward_normal = Vector3d(1, 0, 0);
106        hit.set_face_normal(ray, outward_normal);
107        hit.mat_ptr = mat_ptr;
108        hit.hit_point = ray.at(t);
109        return true;
110    }
111
112    bool bounding_box(double time0, double time1, AABB &output_box) const override {
113        output_box = AABB(Point(x - 0.0001, y0, z0), Point(x + 0.0001, y1, z1));
114        return true;
115    }
116
117     private:
118     double y0{}, y1{}, z0{}, z1{};
119     double x{};
120 };
121
122 #endif //PROJECT_RECTANGLE_H

```

We only define axis-aligned rectangles and rely on transforms to generate other ones. Note that we need to ensure that AABB has no zero length edge by setting thickness to a fixed constant, otherwise intersection detection may fail.

7.1.5 Box (src/hittable/Box.h)

```

1 #include <utility>
2
3 //
4 // Created by meiyixuan on 2021-12-18.
5 //

```

```

6
7 #ifndef PROJECT_BOX_H
8 #define PROJECT_BOX_H
9
10 class Box : public Hittable {
11     public:
12     Box() = default;
13
14     Box(const Point &p0, const Point &p1, const std::shared_ptr<Material> &mat_ptr) : Hittable(nullptr) {
15         // diagonal points
16         min_point = p0;
17         max_point = p1;
18
19         // construct faces
20         faces.add(make_shared<XYRectangle>(p0.x(), p1.x(), p0.y(), p1.y(), p0.z(), mat_ptr));
21         faces.add(make_shared<XYRectangle>(p0.x(), p1.x(), p0.y(), p1.y(), p0.z(), mat_ptr));
22         faces.add(make_shared<XZRectangle>(p0.x(), p1.x(), p0.z(), p1.z(), p0.y(), mat_ptr));
23         faces.add(make_shared<XZRectangle>(p0.x(), p1.x(), p0.z(), p1.z(), p0.y(), mat_ptr));
24         faces.add(make_shared<YZRectangle>(p0.y(), p1.y(), p0.z(), p1.z(), p0.x(), mat_ptr));
25         faces.add(make_shared<YZRectangle>(p0.y(), p1.y(), p0.z(), p1.z(), p0.x(), mat_ptr));
26     }
27
28     bool hit(const Ray &ray, double t_min, double t_max, Hit &hit) const override {
29         return faces.hit(ray, t_min, t_max, hit);
30     }
31
32     bool bounding_box(double time0, double time1, AABB &output_box) const override {
33         output_box = AABB(min_point, max_point);
34         return true;
35     }
36
37     private:
38     Point min_point, max_point;
39     HittableList faces;
40 };
41
42 #endif //PROJECT_BOX_H

```

Box is just six rectangles put together.

7.1.6 Triangle (src/hittable/Triangle.h)

```

1 #include <utility>
2
3 /**
4 // Created by meiyixuan on 2021-12-20.
5 /**
6
7 #ifndef PROJECT_TRIANGLE_H
8 #define PROJECT_TRIANGLE_H
9
10 struct Vertex {
11     Vertex(double _u, double _v, Point _point, Vector3d _normal = {0, 0, 0})
12     : u(_u), v(_v), point(_point), normal(_normal) {}
13
14     double u{0};
15     double v{0};

```

```

16     Point point;
17     Vector3d normal;
18 };
19
20 class Triangle : public Hittable {
21 public:
22     Triangle() = default;
23
24     Triangle(std::shared_ptr<Vertex> v0, std::shared_ptr<Vertex> v1, std::shared_ptr<Vertex> v2,
25               std::shared_ptr<Material> m, std::shared_ptr<BumpMaterial> b = nullptr)
26         : Hittable(std::move(m)), bump_ptr(std::move(b)),
27           normal(normalize(cross(v1->point - v0->point, v2->point - v0->point))),
28           vertices{std::move(v0), std::move(v1), std::move(v2)} {
29         // calculate normal
30         if (vertices[0]->normal.length() <= EPSILON &&
31             vertices[1]->normal.length() <= EPSILON &&
32             vertices[2]->normal.length() <= EPSILON) {
33             use_computed_normal = true;
34         }
35         // calculate uv to xyz map
36         map_uv_2_xyz();
37     }
38
39     bool hit(const Ray &ray, double t_min, double t_max, Hit &hit) const override {
40         // initialize (-MöllerTrumbore intersection algorithm)
41         Vector3d edge1, edge2, h, s, q;
42         double a, f, u, v;
43         edge1 = vertices[1]->point - vertices[0]->point;
44         edge2 = vertices[2]->point - vertices[0]->point;
45         h = cross(ray.direction(), edge2);
46         a = dot(edge1, h);
47
48         // This ray is parallel to this triangle.
49         if (a > -EPSILON && a < EPSILON) return false;
50         f = 1.0 / a;
51         s = ray.origin() - vertices[0]->point;
52         u = f * dot(s, h);
53         if (u < 0.0 || u > 1.0) return false;
54         q = cross(s, edge1);
55         v = f * dot(ray.direction(), q);
56         if (v < 0.0 || u + v > 1.0) return false;
57
58         // At this stage we can compute t to find out where the intersection point is on the line.
59         // basic hit information
60         double t = f * dot(edge2, q);
61         if (t < t_min || t > t_max) return false;
62         hit.t = t;
63         hit.hit_point = ray.at(t);
64         hit.mat_ptr = mat_ptr;
65         // set normal
66         double w0, w1, w2;
67         barycentric(hit.hit_point, w0, w1, w2);
68         if (!use_computed_normal)
69             hit.normal = normalize(w0 * vertices[0]->normal +
70                                   w1 * vertices[1]->normal +
71                                   w2 * vertices[2]->normal);
72         else

```

```

73 hit.normal = normal;
74 // add bump map when available
75 if (bump_ptr) {
76     // get perturbation
77     auto perturb0 = bump_ptr->get_normal(vertices[0]->u, vertices[0]->v);
78     auto perturb1 = bump_ptr->get_normal(vertices[1]->u, vertices[1]->v);
79     auto perturb2 = bump_ptr->get_normal(vertices[2]->u, vertices[2]->v);
80     auto perturb = w0 * perturb0 + w1 * perturb1 + w2 * perturb2;
81     hit.normal = normalize(hit.normal + perturb[0] * u_vec + perturb[1] * v_vec);
82 }
83 hit.set_face_normal(ray, normal);
84 // set u v
85 hit.u = w0 * vertices[0]->u + w1 * vertices[1]->u + w2 * vertices[2]->u;
86 hit.v = w0 * vertices[0]->v + w1 * vertices[1]->v + w2 * vertices[2]->v;
87 return true;
88 }
89
90 bool bounding_box(double time0, double time1, AABB &output_box) const override {
91     // x
92     double x_min = min3(vertices[0]->point.x(), vertices[1]->point.x(), vertices[2]->point.x());
93     double x_max = max3(vertices[0]->point.x(), vertices[1]->point.x(), vertices[2]->point.x());
94     if (x_min == x_max) {
95         x_min -= 0.0001;
96         x_max += 0.0001;
97     }
98
99     // y
100    double y_min = min3(vertices[0]->point.y(), vertices[1]->point.y(), vertices[2]->point.y());
101    double y_max = max3(vertices[0]->point.y(), vertices[1]->point.y(), vertices[2]->point.y());
102    if (y_min == y_max) {
103        y_min -= 0.0001;
104        y_max += 0.0001;
105    }
106
107    // z
108    double z_min = min3(vertices[0]->point.z(), vertices[1]->point.z(), vertices[2]->point.z());
109    double z_max = max3(vertices[0]->point.z(), vertices[1]->point.z(), vertices[2]->point.z());
110    if (z_min == z_max) {
111        z_min -= 0.0001;
112        z_max += 0.0001;
113    }
114
115    output_box = AABB({x_min, y_min, z_min}, {x_max, y_max, z_max});
116    return true;
117 }
118
119 private:
120     bool use_computed_normal{false};
121     Vector3d normal;
122     std::shared_ptr<Vertex> vertices[3];
123     std::shared_ptr<BumpMaterial> bump_ptr;
124     // unit vector of u, v in xyz space
125     Vector3d u_vec, v_vec;
126
127     inline void barycentric(const Point &p, double &u, double &v, double &w) const {
128         // Compute barycentric coordinates (u, v, w) for
129         // point p with respect to triangle (0, 1, 2)

```

```

130 // From Christer Ericson's Real-Time Collision Detection
131 Vector3d v0 = vertices[1]->point - vertices[0]->point;
132 Vector3d v1 = vertices[2]->point - vertices[0]->point;
133 Vector3d v2 = p - vertices[0]->point;
134 double d00 = dot(v0, v0);
135 double d01 = dot(v0, v1);
136 double d11 = dot(v1, v1);
137 double d20 = dot(v2, v0);
138 double d21 = dot(v2, v1);
139 double d = d00 * d11 - d01 * d01;
140 v = (d11 * d20 - d01 * d21) / d;
141 w = (d00 * d21 - d01 * d20) / d;
142 u = 1.0 - v - w;
143 }
144
145 inline void get_triangle_uv(const Point &p, double &u, double &v) const {
146     double a, b, c;
147     barycentric(p, a, b, c);
148     u = a * vertices[0]->u + b * vertices[1]->u + c * vertices[2]->u;
149     v = a * vertices[0]->v + b * vertices[1]->v + c * vertices[2]->v;
150 }
151
152 inline void map_uv_2_xyz() {
153     // 1 wrt 0
154     double a1 = vertices[1]->u - vertices[0]->u;
155     double a2 = vertices[1]->v - vertices[0]->v;
156     Vector3d a_xyz = vertices[1]->point - vertices[0]->point;
157     // 2 wrt 0
158     double b1 = vertices[2]->u - vertices[0]->u;
159     double b2 = vertices[2]->v - vertices[0]->v;
160     Vector3d b_xyz = vertices[2]->point - vertices[0]->point;
161     // denominator and check for degradation
162     double denominator = a1 * b2 - a2 * b1;
163     if (fabs(denominator) < EPSILON) {
164         bool recover_flag = true;
165         // no uv coordinate case
166         if (a1 == 0 && a2 == 0 && b1 == 0 && b2 == 0) return;
167         // u degradation
168         if (vertices[0]->u == vertices[1]->u && vertices[0]->u == vertices[2]->u) {
169             double v_min = min3(vertices[0]->v, vertices[1]->v, vertices[2]->v);
170             double v_max = max3(vertices[0]->v, vertices[1]->v, vertices[2]->v);
171             if (v_min == v_max) recover_flag = false;
172             else {
173                 // we can still recover v in this case
174                 int max_id = v_max == vertices[0]->v ? 0
175                     : v_max == vertices[1]->v ? 1 : 2;
176                 int min_id = v_min == vertices[0]->v ? 0
177                     : v_min == vertices[1]->v ? 1 : 2;
178                 double v_scale = 1 / (v_max - v_min);
179                 v_vec = (vertices[max_id]->point - vertices[min_id]->point) * v_scale;
180                 u_vec = Vector3d();
181                 u_degrade_triangles += 1;
182                 return;
183             }
184         }
185         // v degradation
186         if (vertices[0]->v == vertices[1]->v && vertices[0]->v == vertices[2]->v) {

```

```

187     double u_min = min3(vertices[0]->u, vertices[1]->u, vertices[2]->u);
188     double u_max = max3(vertices[0]->u, vertices[1]->u, vertices[2]->u);
189     if (u_min == u_max) recover_flag = false;
190     else {
191         // we can still recover u in this case
192         int max_id = u_max == vertices[0]->u ? 0
193             : u_max == vertices[1]->u ? 1 : 2;
194         int min_id = u_min == vertices[0]->u ? 0
195             : u_min == vertices[1]->u ? 1 : 2;
196         double u_scale = 1 / (u_max - u_min);
197         v_vec = Vector3d();
198         u_vec = (vertices[max_id]->point - vertices[min_id]->point) * u_scale;
199         v_degrade_triangles += 1;
200         return;
201     }
202 }
203 // unable to recover and log
204 //           std::cerr << "Failed to map uv to xyz" << std::endl;
205 //           if (random_double() < 1) {
206 //               std::cerr << vertices[0]->u << "      " << vertices[0]->v << std::endl;
207 //               std::cerr << vertices[1]->u << "      " << vertices[1]->v << std::endl;
208 //               std::cerr << vertices[1]->u << "      " << vertices[1]->v << std::endl;
209 //               std::cerr << std::endl;
210 //           }
211 u_vec = Vector3d();
212 v_vec = Vector3d();
213 unrecoverable_triangles += 1;
214 return;
215 }
216 // calculate for u
217 double ux = b2 / denominator;
218 double uy = -a2 / denominator;
219 u_vec = ux * a_xyz + uy * b_xyz;
220 // calculate for v
221 double vx = -b1 / denominator;
222 double vy = a1 / denominator;
223 v_vec = vx * a_xyz + vy * b_xyz;
224 normal_triangles += 1;
225 }

226 static inline double min3(const double &a, const double &b, const double &c) {
227     return fmin(fmin(a, b), c);
228 }
229

230 static inline double max3(const double &a, const double &b, const double &c) {
231     return fmax(fmax(a, b), c);
232 }
233
234 };
235
236 #endif //PROJECT_TRIANGLE_H

```

Triangle is much more complicated than other Hittable objects, since it need to support triangular mesh. When initializing a Triangle, we need to compute mapping from uv unit vector to xyz vector. This step is essential if we want to use bump map. Note that many triangles may have degenerated uv coordinates (i.e. vertices having identical uv coordinates) and need to be carefully recovered. Also, surface normal is interpolated using barycentric

coordinates in *hit*.

7.1.7 Mesh (src/hittable/Mesh.h)

```
1 #include <utility>
2
3 //
4 // Created by meiyixuan on 2021-12-20.
5 //
6
7 #ifndef PROJECT_MESH_H
8 #define PROJECT_MESH_H
9
10 class Mesh : public Hittable {
11     public:
12         Mesh() = default;
13
14         explicit Mesh(std::shared_ptr<BVHNode> _mesh) : mesh(std::move(_mesh)) {}
15
16         explicit Mesh(HittableList &list, double time0 = 0, double time1 = 1)
17             : mesh(std::make_shared<BVHNode>(list, time0, time1)) {}
18
19         explicit Mesh(std::vector<std::shared_ptr<Hittable>> &triangles, double time0 = 0, double time1 = 1)
20             : mesh(std::make_shared<BVHNode>(triangles, 0, triangles.size(), time0, time1)) {}
21
22         bool hit(const Ray &ray, double t_min, double t_max, Hit &hit) const override {
23             return mesh->hit(ray, t_min, t_max, hit);
24         }
25
26         bool bounding_box(double time0, double time1, AABB &output_box) const override {
27             return mesh->bounding_box(time0, time1, output_box);
28         }
29
30     private:
31         std::shared_ptr<BVHNode> mesh;
32
33     };
34
35 #endif //PROJECT_MESH_H
```

Mesh is just a BVH of triangles.

7.1.8 Photon Light (src/hittable/PhotonLight.h)

```
1 //
2 // Created by meiyixuan on 2021-12-29.
3 //
4
5 #ifndef PROJECT_PHOTONLIGHT_H
6 #define PROJECT_PHOTONLIGHT_H
7
8 inline Vector3d random_dir(const Vector3d &n);
9
10 class YZRectangleLight : public Hittable {
11     public:
12         YZRectangleLight() = default;
```

```

14 YZRectangleLight(double _y0, double _y1, double _z0, double _z1, double _x, shared_ptr<Material>
15     mat_ptr)
16 : Hittable(std::move(mat_ptr)), y0(_y0), y1(_y1), z0(_z0), z1(_z1), x(_x) {}
17
18 bool hit(const Ray &ray, double t_min, double t_max, Hit &hit) const override {
19     // check if hit plane
20     auto t = (x - ray.origin().x()) / ray.direction().x();
21     if (t < t_min || t > t_max) return false;
22
23     // check if hit rectangle
24     auto y = ray.origin().y() + t * ray.direction().y();
25     auto z = ray.origin().z() + t * ray.direction().z();
26     if (y < y0 || y > y1 || z < z0 || z > z1) return false;
27
28     // hit
29     hit.u = (y - y0) / (y1 - y0);
30     hit.v = (z - z0) / (z1 - z0);
31     hit.t = t;
32     auto outward_normal = Vector3d(1, 0, 0);
33     hit.set_face_normal(ray, outward_normal);
34     hit.mat_ptr = mat_ptr;
35     hit.hit_point = ray.at(t);
36     return true;
37 }
38
39 bool bounding_box(double time0, double time1, AABB &output_box) const override {
40     output_box = AABB(Point(x - 0.0001, y0, z0), Point(x + 0.0001, y1, z1));
41     return true;
42 }
43
44 void generate_photon(Vector3d &origin, Vector3d &direction, double &power) const {
45     origin = Vector3d(y0 + random_double_fixed() * (y1 - y0), x, z0 + random_double_fixed() * (z1 - z0))
46     ;
47     direction = random_dir(Vector3d(1, 0, 0));
48     power = dot(direction, Vector3d(1, 0, 0));
49 }
50
51 private:
52     double y0{}, y1{}, z0{}, z1{};
53     double x{};
54 };
55
56 inline Vector3d random_dir(const Vector3d &n) {
57     Vector3d in_unit_sphere;
58     while (true) {
59         auto p = Vector3d::random_fixed(-1, 1);
60         if (p.squared_length() >= 1) continue;
61         in_unit_sphere = p;
62         break;
63     }
64     if (dot(in_unit_sphere, n) > 0.0)
65         return in_unit_sphere;
66     else
67         return -in_unit_sphere;
68 }
69
70 #endif //PROJECT_PHOTONLIGHT_H

```

Photon Light is derived from normal objects by adding a *generate_photon* function. Since we only need YZ Rectangle lights in our scene, this is done in a hard coded way.

7.2 Complex Models

ObjectParser (src/core/ObjectParser.h)

```

1 #include <utility>
2
3 /**
4 // Created by meiyixuan on 2021-12-20.
5 /**
6
7 #ifndef PROJECT_OBJECTPARSER_H
8 #define PROJECT_OBJECTPARSER_H
9
10 class ObjectParser {
11     public:
12     ObjectParser() = delete;
13
14     explicit ObjectParser(std::string _filename, std::string _mtl_path)
15     : filename(std::move(_filename)), mtl_path(std::move(_mtl_path)) {}
16
17     std::shared_ptr<BVHNode> parse(int bump_map_type = 0, double light_sample_probability = 0,
18     Vector3d sun_dir = Vector3d(),
19     const std::shared_ptr<Material> &default_material
20     = std::make_shared<Lambertian>(Color(1, 1, 1))) {
21         // create reader
22         tinyobj::ObjReaderConfig reader_config;
23         reader_config.mtl_search_path = mtl_path; // Path to material files
24         tinyobj::ObjReader reader;
25         if (!reader.ParseFromFile(filename, reader_config)) {
26             if (!reader.Error().empty()) {
27                 std::cerr << "TinyObjReader: " << reader.Error();
28             }
29             exit(1);
30         }
31         if (!reader.Warning().empty()) {
32             std::cout << "TinyObjReader: " << reader.Warning();
33         }
34
35         // read obj data
36         auto &attrib = reader.GetAttrib();
37         auto &shapes = reader.GetShapes();
38         auto &materials = reader.GetMaterials();
39
40         // build material
41         std::vector<std::shared_ptr<PBRMaterial>> mat_list;
42         std::vector<std::shared_ptr<BumpMaterial>> bump_list;
43         for (const auto &mat: materials) {
44             mat_list.emplace_back(std::make_shared<PBRMaterial>(mat.name, mtl_path,
45             Vector3d(mat.diffuse[0],
46             mat.diffuse[1],
47             mat.diffuse[2]),
48             mat.diffuse_texname,
49             Vector3d(mat.specular[0],
50             mat.specular[1],
51             mat.specular[2])));
52         }
53     }
54 }
```

```

50     mat.specular[1],
51     mat.specular[2]),
52     mat.shininess,
53     mat.specular_texname,
54     Vector3d(mat.emission[0],
55     mat.emission[1],
56     mat.emission[2]),
57     mat.emissive_texname,
58     mat.ior,
59     Vector3d(mat.transmittance[0],
60     mat.transmittance[1],
61     mat.transmittance[2]),
62     mat.dissolve,
63     light_sample_probability,
64     sun_dir));
65     bump_list.emplace_back(mat.bump_texname.empty() ? nullptr :
66     std::make_shared<BumpMaterial>(mtl_path, mat.bump_texname,
67     bump_map_type));
68 }
69
70 // Loop over shapes
71 HittableList mesh_list;
72 for (const auto &shape: shapes) {
73     // initialize storage
74     size_t index_offset = 0;
75     std::vector<std::shared_ptr<Hittable>> triangles;
76
77     // Loop over triangles
78     for (size_t f = 0; f < shape.mesh.num_face_vertices.size(); f++) {
79
80         // check vertex count
81         auto fv = size_t(shape.mesh.num_face_vertices[f]);
82         if (fv != 3) {
83             std::cerr << "A triangle has " << fv << " vertices." << std::endl;
84             index_offset += fv;
85             continue;
86         }
87
88         // construct vertices
89         std::vector<std::shared_ptr<Vertex>> vertex_list;
90         for (size_t v_id = 0; v_id < fv; v_id++) {
91             // access to vertex
92             tinyobj::index_t idx = shape.mesh.indices[index_offset + v_id];
93             tinyobj::real_t vx = attrib.vertices[3 * size_t(idx.vertex_index) + 0];
94             tinyobj::real_t vy = attrib.vertices[3 * size_t(idx.vertex_index) + 1];
95             tinyobj::real_t vz = attrib.vertices[3 * size_t(idx.vertex_index) + 2];
96             Point point(vx, vy, vz);
97
98             // Check if 'normal_index' is zero or positive. negative = no normal data
99             Vector3d normal;
100            if (idx.normal_index >= 0) {
101                tinyobj::real_t nx = attrib.normals[3 * size_t(idx.normal_index) + 0];
102                tinyobj::real_t ny = attrib.normals[3 * size_t(idx.normal_index) + 1];
103                tinyobj::real_t nz = attrib.normals[3 * size_t(idx.normal_index) + 2];
104                normal = Vector3d(nx, ny, nz);
105            }
106        }

```

```

107     // Check if 'texcoord_index' is zero or positive. negative = no texcoord data
108     double u = 0, v = 0;
109     if (idx.texcoord_index >= 0) {
110         tinyobj::real_t tx = attrib.texcoords[2 * size_t(idx.texcoord_index) + 0];
111         tinyobj::real_t ty = attrib.texcoords[2 * size_t(idx.texcoord_index) + 1];
112         u = tx;
113         v = ty;
114     }
115
116     // emplace vertex
117     vertex_list.emplace_back(std::make_shared<Vertex>(u, v, point, normal));
118 }
119 index_offset += fv;
120
121 // per-face material (set to Lambertian for debug)
122 // shape.mesh.material_ids[f];
123 auto mat_id = shape.mesh.material_ids[f];
124 auto mat_ptr = mat_id == -1 ? default_material : mat_list[mat_id];
125 auto bump_ptr = mat_id == -1 ? nullptr : bump_list[shape.mesh.material_ids[f]];
126
127 // create triangle
128 auto triangle_ptr = std::make_shared<Triangle>(vertex_list[0], vertex_list[1],
129 vertex_list[2], mat_ptr, bump_ptr);
130 triangles.push_back(triangle_ptr);
131 }
132
133 // build mesh
134 auto mesh = std::make_shared<Mesh>(triangles);
135 mesh_list.add(mesh);
136 }
137
138 // output parse statistics
139 std::cerr << "*****Parser Statistics*****" << std::endl;
140 std::cerr << "UV Mapping: " << std::endl;
141 std::cerr << "Normal: " << normal_triangles << std::endl;
142 std::cerr << "U-degrade: " << u_degrade_triangles << std::endl;
143 std::cerr << "V-degrade: " << v_degrade_triangles << std::endl;
144 std::cerr << "Unrecoverable: " << unrecoverable_triangles << std::endl;
145 std::cerr << "*****" << std::endl;
146
147 return std::make_shared<BVHNode>(mesh_list, 0.0, 1.0);
148 }
149
150 private:
151 std::string filename;
152 std::string mtl_path;
153 };
154
155 #endif //PROJECT_OBJECTPARSER_H

```

For importing complex models, we use tinyobjloader. It can resolve .obj into three separate lists of vertices, normals and uv coordinates. If the model has a .mtl file that contain material description, tinyobjloader can resolve that into a list of parameters as well. In ObjectParser, we first use material parameters to build a list of PBRMaterials (will be explained in section 8) and then enumerate through all shapes, faces and vertices to build up model mesh.

7.3 Skybox

7.3.1 Base Skybox (src/core/Skybox.h)

```
1 //  
2 // Created by meiyixuan on 2021-12-18.  
3 //  
4  
5 #ifndef PROJECT_SKYBOX_H  
6 #define PROJECT_SKYBOX_H  
7  
8 class Skybox {  
9     public:  
10    virtual Color get_color(Ray r) const = 0;  
11};  
12  
13#endif //PROJECT_SKYBOX_H
```

Rigorously speaking, Skybox is not a hittable object, since it is not derived from Hittable. However, when a ray misses all Hittables in the scene, it automatically collides with the skybox. In this sense, we can view skyboxes as hittables. A skybox takes in a ray and returns color along the ray's direction.

7.3.2 Constant Skybox (src/skybox/ConstantSkybox.h)

```
1 //  
2 // Created by meiyixuan on 2021-12-18.  
3 //  
4  
5 #ifndef PROJECT_CONSTANTSKYBOX_H  
6 #define PROJECT_CONSTANTSKYBOX_H  
7  
8 class ConstantSkybox : public Skybox {  
9     public:  
10    ConstantSkybox() = default;  
11  
12    explicit ConstantSkybox(Color _sky_color) : sky_color(_sky_color) {}  
13  
14    Color get_color(Ray r) const override {  
15        return sky_color;  
16    }  
17  
18    private:  
19    Color sky_color;  
20};  
21  
22#endif //PROJECT_CONSTANTSKYBOX_H
```

A skybox that returns constant color.

7.3.3 Directional Skybox (src/skybox/DirectionalSkybox.h)

```
1 //  
2 // Created by meiyixuan on 2021-12-23.  
3 //  
4  
5 #ifndef PROJECT_DIRECTIONALSKYBOX_H
```

```

6 #define PROJECT_DIRECTIONALSKYBOX_H
7
8 class DirectionalSkybox : public Skybox {
9     public:
10    DirectionalSkybox() = default;
11
12    DirectionalSkybox(double _sky_intensity, double _sun_intensity, Vector3d _dir2sun, double _radius)
13        : sky_intensity(_sky_intensity), sun_intensity(_sun_intensity),
14          dir2sun(normalize(_dir2sun)), sun_radius(_radius) {}
15
16    Color get_color(Ray r) const override {
17        if (pow(fmax(dot(normalize(r.direction()), dir2sun), 0), 3) > sun_radius)
18            return sun_intensity * Color(1, 1, 1);
19        else
20            return sky_intensity * Color(0.5, 0.7, 1.0);
21    }
22
23    private:
24        double sky_intensity{1};
25        double sun_intensity{3};
26        Vector3d dir2sun;
27        double sun_radius{0.6};
28    };
29
30 #endif //PROJECT_DIRECTIONALSKYBOX_H

```

A skybox that simulates sun effect (sun with no atmosphere refraction). Maybe used as a component in TwoLayerSkybox.

7.3.4 Simple Skybox (src/skybox/SimpleSkybox.h)

```

1 //
2 // Created by meiyixuan on 2021-12-18.
3 //
4
5 #ifndef PROJECT_SIMPLESKYBOX_H
6 #define PROJECT_SIMPLESKYBOX_H
7
8 class SimpleSkybox : public Skybox {
9     public:
10    SimpleSkybox() = default;
11
12    explicit SimpleSkybox(double _intensity) : intensity(_intensity) {}
13
14    Color get_color(Ray r) const override {
15        Vector3d unit_direction = normalize(r.direction());
16        auto t = 0.5 * (unit_direction.y() + 1.0);
17        return ((1.0 - t) * Color(1.0, 1.0, 1.0) + t * Color(0.5, 0.7, 1.0))
18            * intensity;
19    }
20
21    private:
22        double intensity{1};
23    };
24
25 #endif //PROJECT_SIMPLESKYBOX_H

```

A skybox that gradually changes from blue to white from top to bottom.

7.3.5 Two Layer Skybox (src/skybox/TwoLayerSkybox.h)

```
1 //  
2 // Created by meiyixuan on 2021-12-28.  
3 //  
4  
5 #ifndef PROJECT_TWOLAYERSKYBOX_H  
6 #define PROJECT_TWOLAYERSKYBOX_H  
7  
8 class TwoLayerSkybox : public Skybox {  
9     public:  
10         TwoLayerSkybox() = default;  
11  
12         TwoLayerSkybox(std::shared_ptr<Skybox> visual, std::shared_ptr<Skybox> GI)  
13             : visual_sky(std::move(visual)), GI_sky(std::move(GI)) {}  
14  
15         Color get_color(Ray r) const override {  
16             if (r.camera_ray()) {  
17                 return visual_sky->get_color(r);  
18             } else {  
19                 return GI_sky->get_color(r);  
20             }  
21         }  
22  
23     private:  
24         std::shared_ptr<Skybox> visual_sky;  
25         std::shared_ptr<Skybox> GI_sky;  
26     };  
27  
28 #endif //PROJECT_TWOLAYERSKYBOX_H
```

A skybox that separates visual effect (i.e. rays that directly hit the skybox) from GI effect.

7.4 Transforms

In ray tracing, for transforms, we generally do not modify real object position. Instead, we modify ray reversely and use the modified ray to do intersection. This method is equivalent to modifying position directly.

7.4.1 Base Transform (src/core/Transform.h)

```
1 //  
2 // Created by meiyixuan on 2021-12-19.  
3 //  
4  
5 #ifndef PROJECT_TRANSFORM_H  
6 #define PROJECT_TRANSFORM_H  
7  
8 class Transform : public Hittable {  
9 };  
10  
11 #endif //PROJECT_TRANSFORM_H
```

Base class for transforms, derived from Hittable.

7.4.2 Translate (src/transform/Translate.h)

```
1 #include <utility>
2
3 //
4 // Created by meiyixuan on 2021-12-19.
5 //
6
7 #ifndef PROJECT_TRANSLATE_H
8 #define PROJECT_TRANSLATE_H
9
10 class Translate : public Transform {
11     public:
12         Translate(std::shared_ptr<Hittable> obj, const Vector3d &_displacement)
13             : obj_ptr(std::move(obj)), displacement(_displacement) {}
14
15     bool hit(const Ray &ray, double t_min, double t_max, Hit &hit) const override {
16         // transform
17         Ray transformed_ray(ray.origin() - displacement, ray.direction(), ray.time());
18
19         // miss
20         if (!obj_ptr->hit(transformed_ray, t_min, t_max, hit))
21             return false;
22
23         // hit
24         hit.hit_point += displacement;
25         hit.set_face_normal(transformed_ray, hit.normal);
26
27         return true;
28     }
29
30     bool bounding_box(double time0, double time1, AABB &output_box) const override {
31         // inf object
32         if (!obj_ptr->bounding_box(time0, time1, output_box)) return false;
33
34         // normal
35         output_box = AABB(output_box.min() + displacement,
36                            output_box.max() + displacement);
37         return true;
38     }
39
40     private:
41         std::shared_ptr<Hittable> obj_ptr;
42         Vector3d displacement;
43     };
44
45 #endif //PROJECT_TRANSLATE_H
```

Translation moves the origin of target ray by $-displacement$.

7.4.3 Rotation (src/transform/RotateY.h)

```
1 #include <utility>
2
3 //
4 // Created by meiyixuan on 2021-12-19.
5 //
```

```

6
7 #ifndef PROJECT_ROTATEY_H
8 #define PROJECT_ROTATEY_H
9
10 class RotateY : public Hittable {
11     public:
12     RotateY(std::shared_ptr<Hittable> p, double angle) : obj_ptr(std::move(p)) {
13         auto radians = deg2rad(angle);
14         sin_theta = sin(radians);
15         cos_theta = cos(radians);
16         has_box = obj_ptr->bounding_box(0, 1, transformed_aabb);
17
18         Point min(-inf, -inf, -inf);
19         Point max(inf, inf, inf);
20
21         for (int i = 0; i < 2; i++) {
22             for (int j = 0; j < 2; j++) {
23                 for (int k = 0; k < 2; k++) {
24                     auto x = i * transformed_aabb.max().x() + (1 - i) * transformed_aabb.min().x();
25                     auto y = j * transformed_aabb.max().y() + (1 - j) * transformed_aabb.min().y();
26                     auto z = k * transformed_aabb.max().z() + (1 - k) * transformed_aabb.min().z();
27                     auto new_x = cos_theta * x + sin_theta * z;
28                     auto new_z = -sin_theta * x + cos_theta * z;
29                     Vector3d tester(new_x, y, new_z);
30                     for (int c = 0; c < 3; c++) {
31                         min[c] = fmin(min[c], tester[c]);
32                         max[c] = fmax(max[c], tester[c]);
33                     }
34                 }
35             }
36         }
37         transformed_aabb = AABB(min, max);
38     }
39
40     bool hit(const Ray &ray, double t_min, double t_max, Hit &hit) const override {
41         // generate transformed ray
42         auto origin = ray.origin();
43         auto direction = ray.direction();
44         origin[0] = cos_theta * ray.origin()[0] - sin_theta * ray.origin()[2];
45         origin[2] = sin_theta * ray.origin()[0] + cos_theta * ray.origin()[2];
46         direction[0] = cos_theta * ray.direction()[0] - sin_theta * ray.direction()[2];
47         direction[2] = sin_theta * ray.direction()[0] + cos_theta * ray.direction()[2];
48         Ray transformed_ray(origin, direction, ray.time());
49
50         // miss
51         if (!obj_ptr->hit(transformed_ray, t_min, t_max, hit)) return false;
52
53         // hit
54         auto p = hit.hit_point;
55         auto normal = hit.normal;
56         p[0] = cos_theta * hit.hit_point[0] + sin_theta * hit.hit_point[2];
57         p[2] = -sin_theta * hit.hit_point[0] + cos_theta * hit.hit_point[2];
58         normal[0] = cos_theta * hit.normal[0] + sin_theta * hit.normal[2];
59         normal[2] = -sin_theta * hit.normal[0] + cos_theta * hit.normal[2];
60         hit.hit_point = p;
61         hit.set_face_normal(transformed_ray, normal);
62     }

```

```

63     return true;
64 }
65
66 bool bounding_box(double time0, double time1, AABB &output_box) const override {
67     output_box = transformed_aabb;
68     return has_box;
69 }
70
71 private:
72 std::shared_ptr<Hittable> obj_ptr;
73 double sin_theta{0}, cos_theta{1};
74 bool has_box;
75 AABB transformed_aabb;
76 };
77
78 #endif //PROJECT_ROTATEY_H

```

We does not implement general rotation using matrix multiplication. Instead, we only implement around axis rotation, using code similar to that of Ray Tracing Mini-books.

8 Materials & Textures

8.1 Materials

8.1.1 Base Material (src/core/Material.h)

```

1 /**
2 // Created by meiyixuan on 2021-12-11.
3 /**
4
5 #ifndef PROJECT_MATERIAL_H
6 #define PROJECT_MATERIAL_H
7
8 class Hit;
9
10 class Material {
11 public:
12     virtual bool scatter(const Ray &ray_in, Hit &hit, Ray &scattered_ray) const = 0;
13
14     virtual Color brdf(const Ray &ray_in, const Ray &ray_out, const Hit &hit) const = 0;
15
16     virtual Color emit(double u, double v, const Point &p) const {
17         return {0, 0, 0};
18     };
19
20 };
21
22 #endif //PROJECT_MATERIAL_H

```

This is base class for materials. Every material has a *scatter* function that computes scattered rays (i.e. reflection, refraction, etc.), a *brdf* function that computes its response to illumination and a *emit* function that computes self-emission.

8.1.2 Lambertian (src/material/Lambertian.h)

```

1 // 
2 // Created by meiyixuan on 2021-12-18.
3 //
4
5 #ifndef PROJECT_LAMBERTIAN_H
6 #define PROJECT_LAMBERTIAN_H
7
8 class Lambertian : public Material {
9     public:
10    explicit Lambertian(const Color &a) : albedo(std::make_shared<ColorTexture>(a)) {}
11
12    explicit Lambertian(std::shared_ptr<Texture> a) : albedo(std::move(a)) {}
13
14    bool scatter(const Ray &ray_in, Hit &hit, Ray &scattered_ray) const override {
15        // get scatter direction
16        auto scatter_direction = hit.normal + random_unit_vector();
17        if (scatter_direction.near_zero())
18            scatter_direction = hit.normal;
19
20        // generate scattered rays
21        scattered_ray = Ray(hit.hit_point, scatter_direction, ray_in.time());
22        hit.scatter_mode = 0;
23        return true;
24    }
25
26    Color brdf(const Ray &ray_in, const Ray &ray_out, const Hit &hit) const override {
27        return albedo->uv_color(hit.u, hit.v, hit.hit_point);
28    }
29
30    private:
31    std::shared_ptr<Texture> albedo;
32};
33
34 #endif //PROJECT_LAMBERTIAN_H

```

Lambertian is a material that simulate matte, i.e. complete diffuse surface. It scatter rays with direction that follows cos probability distribution. The BRDF function is just the material's albedo.

8.1.3 Metal (src/material/Metal.h)

```

1 #include <utility>
2
3 //
4 // Created by meiyixuan on 2021-12-18.
5 //
6
7 #ifndef PROJECT_METAL_H
8 #define PROJECT_METAL_H
9
10 class Metal : public Material {
11     public:
12        Metal(const Color &a, double f) : albedo(std::make_shared<ColorTexture>(a)), fuzz(f < 1 ? f : 1) {}
13
14        Metal(std::shared_ptr<Texture> a, double f) : albedo(std::move(a)), fuzz(f < 1 ? f : 1) {}
15
16        bool scatter(const Ray &ray_in, Hit &hit, Ray &scattered_ray) const override {
17            // get reflected direction (include perturbation for imperfect reflection)

```

```

18 Vector3d reflected_dir = reflect(normalize(ray_in.direction()), hit.normal);
19 reflected_dir += fuzz * random_in_unit_sphere();
20
21 // generate rays
22 scattered_ray = Ray(hit.hit_point, reflected_dir, ray_in.time());
23 hit.scatter_mode = 1;
24 return (dot(reflected_dir, hit.normal) > 0);
25 }
26
27 Color brdf(const Ray &ray_in, const Ray &ray_out, const Hit &hit) const override {
28     return albedo->uv_color(hit.u, hit.v, hit.hit_point);
29 }
30
31 private:
32 std::shared_ptr<Texture> albedo;
33 double fuzz;
34 };
35
36 #endif //PROJECT_METAL_H

```

Metal simulates complete reflection surface. We can use *fuzz* to control reflection perfectness. *fuzz* = 0 means perfect reflection and *fuzz* = 1 means very fuzzy imperfect reflection.

8.1.4 Dielectric (src/material/Dielectric.h)

```

1 //
2 // Created by meiyixuan on 2021-12-18.
3 //
4
5 #ifndef PROJECT_DIELECTRIC_H
6 #define PROJECT_DIELECTRIC_H
7
8 class Dielectric : public Material {
9     public:
10     explicit Dielectric(double index_of_refraction, Color tf = Vector3d(1, 1, 1))
11         : ir(index_of_refraction), transmission_filter(tf) {}
12
13     bool scatter(const Ray &ray_in, Hit &hit, Ray &scattered_ray) const override {
14         // calculate parameters
15         double refraction_ratio = hit.front_face ? (1.0 / ir) : ir;
16
17         Vector3d unit_direction = normalize(ray_in.direction());
18         double cos_theta = fmin(dot(-unit_direction, hit.normal), 1.0);
19         double sin_theta = sqrt(1.0 - cos_theta * cos_theta);
20
21         // generate new ray's direction
22         bool cannot_refract = refraction_ratio * sin_theta > 1.0;
23         Vector3d direction;
24         if (cannot_refract || reflectance(cos_theta, refraction_ratio) > random_double()) {
25             direction = reflect(unit_direction, hit.normal);
26             hit.scatter_mode = 1;
27         } else {
28             direction = refract(unit_direction, hit.normal, refraction_ratio);
29             hit.scatter_mode = 2;
30         }
31
32     }

```

```

33     scattered_ray = Ray(hit.hit_point, direction, ray_in.time());
34     return true;
35 }
36
37 Color brdf(const Ray &ray_in, const Ray &ray_out, const Hit &hit) const override {
38     return transmission_filter;
39 }
40
41 private:
42 // eta of this material (index of refraction)
43 double ir;
44 Color transmission_filter{1.0, 1.0, 1.0};
45
46 static double reflectance(double cosine, double ref_idx) {
47     // Use Schlick's approximation for reflectance.
48     auto r0 = (1 - ref_idx) / (1 + ref_idx);
49     r0 = r0 * r0;
50     return r0 + (1 - r0) * pow((1 - cosine), 5);
51 }
52 };
53
54 #endif //PROJECT_DIELECTRIC_H

```

Dielectric simulates translucent materials. Light in general refracts at the surface. However, if the input ray grazes the surface, i.e. the direction is almost parallel to the surface, reflection, instead of refraction, will happen. Dielectric also simulates this effect in out implementation.

8.1.5 PBR Material (src/material/PBRMaterial.h)

```

1 #include <utility>
2
3 //
4 // Created by meiyixuan on 2021-12-22.
5 //
6
7 #ifndef PROJECT_PBRMATERIAL_H
8 #define PROJECT_PBRMATERIAL_H
9
10 class PBRMaterial : public Material {
11     public:
12     PBRMaterial(std::string name, const std::string &mat_pth,
13     Color _kd, const std::string &diffuse_map_name,
14     Color _ks, double _shininess, const std::string &specular_map_name,
15     Color _ke, const std::string &emission_map_name,
16     double _ir, Color _tf, double _dissolve,
17     double _light_sample_probability = 0, Vector3d _sun_dir = Vector3d())
18     : material_name(std::move(name)),
19     kd(_kd),
20     diffuse_texture(diffuse_map_name.empty() ? nullptr :
21     std::make_shared<ImageTexture>((mat_pth + diffuse_map_name).c_str())),
22     ks(_ks), shininess(_shininess),
23     specular_texture(specular_map_name.empty() ? nullptr :
24     std::make_shared<ImageTexture>((mat_pth + specular_map_name).c_str())),
25     ke(_ke), emission_texture(emission_map_name.empty() ? nullptr :
26     std::make_shared<ImageTexture>((mat_pth + emission_map_name).c_str())),
27     ir(_ir), transmission_filter(_tf), dissolve(_dissolve),
28     sample_light_prob(_light_sample_probability), sun_dir(_sun_dir) {

```

```

29     auto kd_len = kd.squared_length();
30     auto ks_len = ks.squared_length();
31     auto total_energy = kd_len + ks_len + EPSILON;
32     prob_refract = 1 - dissolve;
33     prob_diffuse = dissolve * kd_len / total_energy;
34     prob_specular = dissolve * ks_len / total_energy;
35 }
36
37 Color emit(double u, double v, const Point &p) const override {
38     return ke * (emission_texture ? emission_texture->uv_color(u, v, p) : Color(1, 1, 1));
39 }
40
41 bool scatter(const Ray &ray_in, Hit &hit, Ray &scattered_ray) const override {
42     auto mode = scatter_type();
43     if (mode == 0) {
44         // mode = 0: defuse material
45         if (hit.remaining_bounce == 1) {
46             // sample to light direction at last bounce
47             // get scatter direction
48             auto scatter_direction = sun_dir;
49             if (dot(scatter_direction, hit.normal) <= 0)
50                 scatter_direction = hit.normal + random_unit_vector();
51             if (scatter_direction.near_zero())
52                 scatter_direction = hit.normal;
53             // generate scattered rays
54             scattered_ray = Ray(hit.hit_point, scatter_direction, ray_in.time());
55             hit.scatter_mode = 0;
56             return true;
57         }
58         if (random_double() > sample_light_prob) {
59             // a random direction sampler
60             // get scatter direction
61             auto scatter_direction = hit.normal + random_unit_vector();
62             if (scatter_direction.near_zero())
63                 scatter_direction = hit.normal;
64             // generate scattered rays
65             scattered_ray = Ray(hit.hit_point, scatter_direction, ray_in.time());
66             hit.scatter_mode = 0;
67             return true;
68         } else {
69             // sample more light for better convergence
70             // get scatter direction
71             auto scatter_direction = sun_dir + 0.02 * random_in_unit_sphere();
72             if (dot(scatter_direction, hit.normal) <= 0)
73                 scatter_direction = hit.normal + random_unit_vector();
74             if (scatter_direction.near_zero())
75                 scatter_direction = hit.normal;
76             // generate scattered rays
77             scattered_ray = Ray(hit.hit_point, scatter_direction, ray_in.time());
78             hit.scatter_mode = 0;
79             return true;
80         }
81     } else if (mode == 1) {
82         // get reflected direction (include perturbation for imperfect reflection)
83         Vector3d reflected_dir = reflect(normalize(ray_in.direction()), hit.normal);
84         // reflected_dir += random_in_unit_sphere();
85     }

```

```

86     // generate rays
87     scattered_ray = Ray(hit.hit_point, reflected_dir, ray_in.time());
88     hit.scatter_mode = 1;
89     return (dot(reflected_dir, hit.normal) > 0);
90 } else if (mode == 2) {
91     // calculate parameters
92     double refraction_ratio = hit.front_face ? (1.0 / ir) : ir;
93
94     Vector3d unit_direction = normalize(ray_in.direction());
95     double cos_theta = fmin(dot(-unit_direction, hit.normal), 1.0);
96     double sin_theta = sqrt(1.0 - cos_theta * cos_theta);
97
98     // generate new ray's direction
99     bool cannot_refract = refraction_ratio * sin_theta > 1.0;
100    Vector3d direction;
101    if (cannot_refract || reflectance(cos_theta, refraction_ratio) > random_double()) {
102        direction = reflect(unit_direction, hit.normal);
103        hit.scatter_mode = 1;
104    } else {
105        direction = refract(unit_direction, hit.normal, refraction_ratio);
106        hit.scatter_mode = 2;
107    }
108    scattered_ray = Ray(hit.hit_point, direction, ray_in.time());
109    return true;
110 } else {
111     std::cerr << "Unknown scatter type!" << std::endl;
112     return false;
113 }
114 }

115
116 inline Color brdf(const Ray &ray_in, const Ray &ray_out, const Hit &hit) const override {
117     // Phong
118     if (hit.scatter_mode == 0 || hit.scatter_mode == 1) {
119         auto diffuse_color = diffuse_texture ? kd * diffuse_texture->uv_color(hit.u, hit.v, hit.hit_point)
120             : kd;
121         auto specular_color_base = specular_texture ? ks * specular_texture->uv_color(hit.u, hit.v, hit.
122             hit_point)
123             : ks;
124         Vector3d reflected_dir = reflect(normalize(ray_in.direction()), hit.normal);
125         auto specular_exponent = pow(dot(reflected_dir, normalize(ray_out.direction())), shininess)
126             / dot(hit.normal, -normalize(ray_in.direction())));
127         specular_exponent = fmax(EPSILON * EPSILON, specular_exponent);
128         specular_exponent = fmin(5.0, specular_exponent);
129         //           if (random_double() < 0.0000001)
130         //               std::cerr << "specular_exponent " << specular_exponent << std::endl;
131
132         return diffuse_color + specular_color_base * specular_exponent;
133     } else if (hit.scatter_mode == 2) {
134         return transmission_filter;
135     } else {
136         std::cerr << "Unknown scatter mode" << std::endl;
137         return {0, 1, 1};
138     }
139 }
140
141 private:
142     // basic property

```

```

142 std::string material_name;
143
144 // diffuse
145 Color kd;
146 std::shared_ptr<Texture> diffuse_texture;
147 double prob_diffuse;
148
149 // specular
150 Color ks;
151 double shininess; // specular exponent
152 std::shared_ptr<Texture> specular_texture;
153 double prob_specular;
154
155 // refraction
156 double ir; // dielectric's eta of refraction
157 Color transmission_filter; // attenuation to refracted ray
158 double dissolve;
159 double prob_refract;
160
161 // emission
162 Color ke;
163 std::shared_ptr<Texture> emission_texture;
164
165 // sampler
166 double sample_light_prob{0};
167 Vector3d sun_dir;
168
169 int scatter_type() const {
170     // 0 diffuse, 1 reflect, 2 refract
171     auto val = random_double();
172     if (val <= prob_diffuse) return 0;
173     else if (val <= prob_diffuse + prob_specular) return 1;
174     else return 2;
175 }
176
177 static double reflectance(double cosine, double ref_idx) {
178     // Use Schlick's approximation for reflectance.
179     auto r0 = (1 - ref_idx) / (1 + ref_idx);
180     r0 = r0 * r0;
181     return r0 + (1 - r0) * pow((1 - cosine), 5);
182 }
183
184 };
185
186 #endif //PROJECT_PBRMATERIAL_H

```

PBR material is a material that have diffuse, reflection and refraction. It also supports image textures for every scatter mode. The lighting model is Phong. PBRMaterial is supposed to be used when importing materials from .mtl files.

8.1.6 Isotropic (src/material/Isotropic.h)

```

1 #include <utility>
2
3 //
4 // Created by meiyixuan on 2021-12-19.
5 //

```

```

6
7 #ifndef PROJECT_ISOTROPIC_H
8 #define PROJECT_ISOTROPIC_H
9
10 class Isotropic : public Material {
11     public:
12         explicit Isotropic(Color color) : albedo(std::make_shared<ColorTexture>(color)) {}
13
14         explicit Isotropic(std::shared_ptr<Texture> texture) : albedo(std::move(texture)) {}
15
16         bool scatter(const Ray &ray_in, Hit &hit, Ray &scattered_ray) const override {
17             scattered_ray = Ray(hit.hit_point, random_in_unit_sphere(), ray_in.time());
18             hit.scatter_mode = 3;
19             return true;
20         }
21
22         Color brdf(const Ray &ray_in, const Ray &ray_out, const Hit &hit) const override {
23             return albedo->uv_color(hit.u, hit.v, hit.hit_point);
24         }
25
26     private:
27         std::shared_ptr<Texture> albedo;
28     };
29
30 #endif //PROJECT_ISOTROPIC_H

```

Isotropic material is used in participating media for volumetric rendering. It scatters the input ray to a random direction (i.e. isotropically).

8.1.7 Diffuse Light (src/light/DiffuseLight.h)

```

1 //
2 // Created by meiyixuan on 2021-12-18.
3 //
4
5 #ifndef PROJECT_DIFFUSELIGHT_H
6 #define PROJECT_DIFFUSELIGHT_H
7
8 class DiffuseLight : public Light {
9     public:
10         explicit DiffuseLight(std::shared_ptr<Texture> texture, double _intensity = 1)
11             : emit_texture(std::move(texture)), intensity(_intensity) {}
12
13         explicit DiffuseLight(Color emit_color, double _intensity = 1) :
14             emit_texture(std::make_shared<ColorTexture>(emit_color)),
15             intensity(_intensity) {}
16
17         bool scatter(const Ray &ray_in, Hit &hit, Ray &scattered_ray) const override {
18             // lights don't scatter
19             return false;
20         }
21
22         Color brdf(const Ray &ray_in, const Ray &ray_out, const Hit &hit) const override {
23             // no brdf
24             return {0, 0, 0};
25         }
26

```

```

27 Color emit(double u, double v, const Point &p) const override {
28     return emit_texture->uv_color(u, v, p) * intensity;
29 }
30
31 private:
32 std::shared_ptr<Texture> emit_texture;
33 double intensity{1};
34 };
35
36 #endif //PROJECT_DIFFUSELIGHT_H

```

This material simulates a light that emits according to *emit_texture*.

8.1.8 Bump Material (src/material/BumpMaterial.h)

```

1 // 
2 // Created by meiyixuan on 2021-12-25.
3 //
4
5 #ifndef PROJECT_BUMPMATERIAL_H
6 #define PROJECT_BUMPMATERIAL_H
7
8 class BumpMaterial : public Material {
9     public:
10     BumpMaterial() = default;
11
12     BumpMaterial(const std::string &mat_pth, const std::string &bump_map_name, int bump_map_type)
13         : bump_map(bump_map_name.empty() ? nullptr :
14             std::make_shared<BumpTexture>((mat_pth + bump_map_name).c_str(),
15             bump_map_type)) {}
16
17     bool scatter(const Ray &ray_in, Hit &hit, Ray &scattered_ray) const override { return false; }
18
19     Color brdf(const Ray &ray_in, const Ray &ray_out, const Hit &hit) const override { return {}; }
20
21     Vector3d get_normal(double u, double v) {
22         return bump_map->get_normal(u, v);
23     }
24
25     private:
26     std::shared_ptr<BumpTexture> bump_map;
27 };
28
29 #endif //PROJECT_BUMPMATERIAL_H

```

Rigorously speaking, bump material is not a material, since it is not used for rendering. This material contains a *bump_map* texture that is applied to the surface normal of models. It changes surface normals and adds detail without modifying model geometry. For detailed implementation of bump map, see section 8.2.

8.2 Textures

8.2.1 Base Texture (src/core/Texture.h)

```

1 // 
2 // Created by meiyixuan on 2021-12-18.
3 //

```

```

4
5 #ifndef PROJECT_TEXTURE_H
6 #define PROJECT_TEXTURE_H
7
8 class Texture {
9 public:
10    virtual Color uv_color(double u, double v, const Point &p) const = 0;
11 };
12
13 #endif //PROJECT_TEXTURE_H

```

Base class for textures. A texture takes in uv-coordinates and hit point and returns color at that point.

8.2.2 Color Texture (src/texture/ColorTexture.h)

```

1 //
2 // Created by meiyixuan on 2021-12-18.
3 //
4
5 #ifndef PROJECT_COLORTEXTURE_H
6 #define PROJECT_COLORTEXTURE_H
7
8 class ColorTexture : public Texture {
9 public:
10    ColorTexture() = default;
11
12    explicit ColorTexture(Color c) : texture_color(c) {}
13
14    ColorTexture(double r, double g, double b) : texture_color{r, g, b} {}
15
16    Color uv_color(double u, double v, const Point &p) const override {
17        return texture_color;
18    }
19
20 private:
21    Color texture_color;
22};
23
24 #endif //PROJECT_COLORTEXTURE_H

```

A texture of pure color.

8.2.3 Checker Texture (src/texture/CheckerTexture.h)

```

1 //
2 // Created by meiyixuan on 2021-12-18.
3 //
4
5 #ifndef PROJECT_CHECKERTEXTURE_H
6 #define PROJECT_CHECKERTEXTURE_H
7
8 class CheckerTexture : public Texture {
9 public:
10    CheckerTexture() = default;
11
12    CheckerTexture(std::shared_ptr<Texture> _odd, std::shared_ptr<Texture> _even, double _step_size = 10)
13        : odd(std::move(_odd)), even(std::move(_even)), step_size(_step_size) {}

```

```

14
15 CheckerTexture(const Color &_odd, const Color &_even, double _step_size = 10)
16 : odd(std::make_shared<ColorTexture>(_odd)),
17 even(std::make_shared<ColorTexture>(_even)),
18 step_size(_step_size) {}
19
20 Color uv_color(double u, double v, const Point &p) const override {
21     auto sines = sin(step_size * p.x()) * sin(step_size * p.y()) * sin(step_size * p.z());
22     if (sines < 0)
23         return odd->uv_color(u, v, p);
24     else
25         return even->uv_color(u, v, p);
26 }
27
28 private:
29     double step_size{10};
30     std::shared_ptr<Texture> odd;
31     std::shared_ptr<Texture> even;
32 };
33
34 #endif //PROJECT_CHECKERTEXTURE_H

```

A simple procedural texture that simulates checker board.

8.2.4 Perlin Texture (src/texture/PerlinTexture.h)

```

1 //
2 // Created by meiyixuan on 2021-12-18.
3 //
4
5 #ifndef PROJECT_PERLINTEXTURE_H
6 #define PROJECT_PERLINTEXTURE_H
7
8 class PerlinTexture : public Texture {
9     public:
10     PerlinTexture() = default;
11
12     explicit PerlinTexture(double _scale) : scale(_scale) {}
13
14     Color uv_color(double u, double v, const Point &p) const override {
15         // if we use "Color(1,1,1) * 0.5 * (1.0 + noise.noise(scale * p))",
16         // we will get a lighter looking Perlin texture.
17         return Color(1, 1, 1) * noise.turbulence(scale * p);
18     }
19
20     private:
21     // scale = 1: very low freq
22     // scale = 4: moderate frequency
23     PerlinNoise noise;
24     double scale{1};
25 };
26
27 #endif //PROJECT_PERLINTEXTURE_H

```

Perlin Texture is generated directly from Perlin noise.

8.2.5 Marble Texture (src/texture/MarbleTexture.h)

```
1 //  
2 // Created by meiyixuan on 2021-12-18.  
3 //  
4  
5 #ifndef PROJECT_MARBLETEXTURE_H  
6 #define PROJECT_MARBLETEXTURE_H  
7  
8 class MarbleTexture : public Texture {  
9     public:  
10         MarbleTexture() = default;  
11  
12         explicit MarbleTexture(double _scale) : scale(_scale) {}  
13  
14         Color uv_color(double u, double v, const Point &p) const override {  
15             // This will render a marble texture.  
16             return Color(1, 1, 1) * 0.5 * (1 + sin(scale * p.z() + 10 * noise.turbulence(scale * p / 4)));  
17         }  
18  
19     private:  
20         PerlinNoise noise;  
21         double scale{4};  
22     };  
23  
24 #endif //PROJECT_MARBLETEXTURE_H
```

A very nice example of procedural texture. It is similar to Perlin Texture in that they both use Perlin noise. However, this texture is insensitive to scaling and can simulate marble on any surface.

8.2.6 Image Texture (src/texture/ImageTexture.h)

```
1 //  
2 // Created by meiyixuan on 2021-12-18.  
3 //  
4  
5 #ifndef PROJECT_IMAGETEXTURE_H  
6 #define PROJECT_IMAGETEXTURE_H  
7  
8 class ImageTexture : public Texture {  
9     public:  
10         const static int bytes_per_pixel = 3;  
11         constexpr static double color_scale = 1.0 / 255.0;  
12  
13         ImageTexture() = default;  
14  
15         explicit ImageTexture(const char *filename, int clamp_type = 1) {  
16             // load image using stb_image  
17             auto components_per_pixel = bytes_per_pixel;  
18             data = stbi_load(filename, &width, &height, &components_per_pixel, components_per_pixel);  
19  
20             // check integrity  
21             if (!data) {  
22                 std::cerr << "ERROR: Could not load texture image file " << filename << ".\n";  
23                 width = height = 0;  
24             }  
25         }
```

```

26 // set parameter
27 bytes_per_scanline = bytes_per_pixel * width;
28 uv_clamp_type = clamp_type;
29 }
30
31 ~ImageTexture() { delete[] data; }
32
33 Color uv_color(double u, double v, const Vector3d &p) const override {
34     // return cyan if no texture image found
35     if (data == nullptr) return {0, 1, 1};
36
37     // parse input
38     if (uv_clamp_type == 0) {
39         u = clamp(u, 0.0, 1.0);
40         v = 1.0 - clamp(v, 0.0, 1.0);
41     } else if (uv_clamp_type == 1) {
42         u = u - floor(u);
43         v = v - floor(v);
44     }
45     if (u < 0 || u > 1.0) std::cerr << "error u " << u << std::endl;
46     if (v < 0 || v > 1.0) std::cerr << "error v " << v << std::endl;
47     auto i = static_cast<int>(u * width);
48     auto j = static_cast<int>(v * height);
49     i = i >= width ? width - 1 : i;
50     j = j >= height ? height - 1 : j;
51
52     // locate pixel and return
53
54     auto pixel = data + j * bytes_per_scanline + i * bytes_per_pixel;
55     return {color_scale * pixel[0], color_scale * pixel[1], color_scale * pixel[2]};
56 }
57
58 private:
59     unsigned char *data{nullptr};
60     int width{0}, height{0};
61     int bytes_per_scanline{0};
62     int uv_clamp_type{-1}; // 0: clamp 1: mod
63 };
64
65 #endif //PROJECT_IMAGETEXTURE_H

```

Image texture is implemented using stb_image. Its return value depends on the attached image. It is used in PBRMaterial as default textures.

8.2.7 Bump Texture (src/texture/BumpTexture.h)

```

1 //
2 // Created by meiyixuan on 2021-12-25.
3 //
4
5 #ifndef PROJECT_BUMPTEXTURE_H
6 #define PROJECT_BUMPTEXTURE_H
7
8 class BumpTexture : public Texture {
9     public:
10     const static int bytes_per_pixel = 3;
11     constexpr static double color_scale = 1.0 / 255.0;

```

```

12
13 BumpTexture() = default;
14
15 explicit BumpTexture(const char *filename, int _bump_map_type, double _bump_scale = 0.01,
16 int clamp_type = 1) {
17     // load image using stb image
18     auto components_per_pixel = bytes_per_pixel;
19     data = stbi_load(filename, &width, &height, &components_per_pixel, components_per_pixel);
20
21     // check integrity
22     if (!data) {
23         std::cerr << "ERROR: Could not load texture image file '" << filename << "'.\n";
24         width = height = 0;
25     }
26
27     // set parameter
28     bytes_per_scanline = bytes_per_pixel * width;
29     uv_clamp_type = clamp_type;
30     bump_scale = _bump_scale;
31     bump_map_type = _bump_map_type;
32 }
33
34 ~BumpTexture() { delete [] data; }
35
36 Color uv_color(double u, double v, const Vector3d &p) const override { return {}; }
37
38 Vector3d get_normal(double u, double v) {
39     // return 0 if bump texture not found
40     if (data == nullptr) return {0, 0, 1};
41
42     // parse input
43     if (uv_clamp_type == 0) {
44         u = clamp(u, 0.0, 1.0);
45         v = 1.0 - clamp(v, 0.0, 1.0);
46     } else if (uv_clamp_type == 1) {
47         u = u - floor(u);
48         v = v - floor(v);
49     }
50     if (u < 0 || u > 1.0) std::cerr << "error u " << u << std::endl;
51     if (v < 0 || v > 1.0) std::cerr << "error v " << v << std::endl;
52     auto i = static_cast<int>(u * width);
53     auto j = static_cast<int>(v * height);
54     i = i >= width ? width - 1 : i;
55     j = j >= height ? height - 1 : j;
56
57     // return based on type
58     if (bump_map_type == 0) {
59         // height map
60         // calculate adjacent coordinates
61         int ip1 = i + 1 == width ? 0 : i + 1;
62         int im1 = i - 1 == -1 ? width - 1 : i - 1;
63         int jp1 = j + 1 == height ? 0 : j + 1;
64         int jm1 = j - 1 == -1 ? height - 1 : j - 1;
65
66         // calculate normal
67         Vector3d normal = Vector3d(-bump_scale * (h(ip1, j) - h(im1, j)),
68             -bump_scale * (h(i, jp1) - h(i, jm1)),

```

```

69     1);
70     return normal;
71 } else if (bump_map_type == 1) {
72     // normal map
73     auto pixel = data + j * bytes_per_scanline + i * bytes_per_pixel;
74     return {color_scale * pixel[0], color_scale * pixel[1], color_scale * pixel[2]};
75 } else if (bump_map_type == 2) {
76     // auto type detection
77     auto pixel = data + j * bytes_per_scanline + i * bytes_per_pixel;
78     if (pixel[0] == pixel[1] && pixel[0] == pixel[2]) {
79         // height map
80         // calculate adjacent coordinates
81         int ip1 = i + 1 == width ? 0 : i + 1;
82         int im1 = i - 1 == -1 ? width - 1 : i - 1;
83         int jp1 = j + 1 == height ? 0 : j + 1;
84         int jm1 = j - 1 == -1 ? height - 1 : j - 1;
85
86         // calculate normal
87         Vector3d normal = Vector3d(-bump_scale * (h(ip1, j) - h(im1, j)),
88             -bump_scale * (h(i, jp1) - h(i, jm1)),
89             1);
90         return normal;
91     } else {
92         // normal map
93         return {color_scale * pixel[0], color_scale * pixel[1], color_scale * pixel[2]};
94     }
95 } else {
96     std::cerr << "Unknown bump map type" << std::endl;
97     return {0, 0, 1};
98 }
99 }

100 private:
101     unsigned char *data{nullptr};
102     int width{0}, height{0};
103     int bytes_per_scanline{0};
104     int uv_clamp_type{-1}; // 0: clamp 1: mod
105     double bump_scale{0.01};
106     int bump_map_type{-1}; // 0: height, 1: normal, 2: auto
107
108     inline int h(int i, int j) {
109         return (data + j * bytes_per_scanline + i * bytes_per_pixel)[0];
110     }
111 };
112 };
113
114 #endif //PROJECT_BUMPITEXTURE_H

```

Bump texture stores perturbation to surface normal. When we talk about bump map, we may either refer to normal map (i.e. the texture stores perturbation along three axes in its three channels) or height map (i.e. the texture only uses one channel to store height information, normal perturbation needs to be calculated by differentiation). Also, for uv coordinate larger than one, there are two different interpretations: clamp (i.e. 0-1 clamp) or mod (take $u - \text{floor}(u)$ as real u). In bump map we deal with these cases separately for compatibility with all kinds of models.

9 Visual Effects

9.1 Camera Effects

10 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>)