Ray Tracing Renderer

Mei Yixuan

January 9, 2022

1 Introduction

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

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

2 Results

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

2.1 Hollow Glass Ball

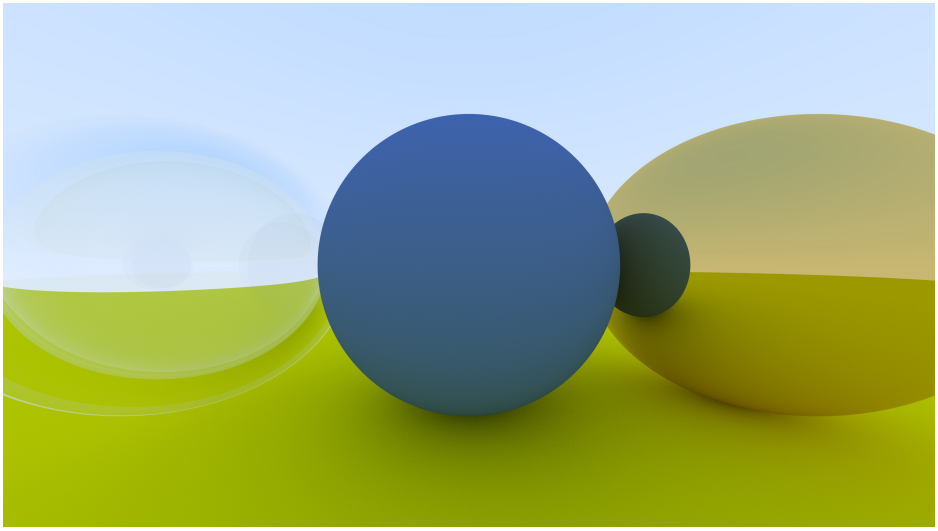

Figure 1: Hollow Glass Ball

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

2.2 Hollow Glass Ball Small FOV

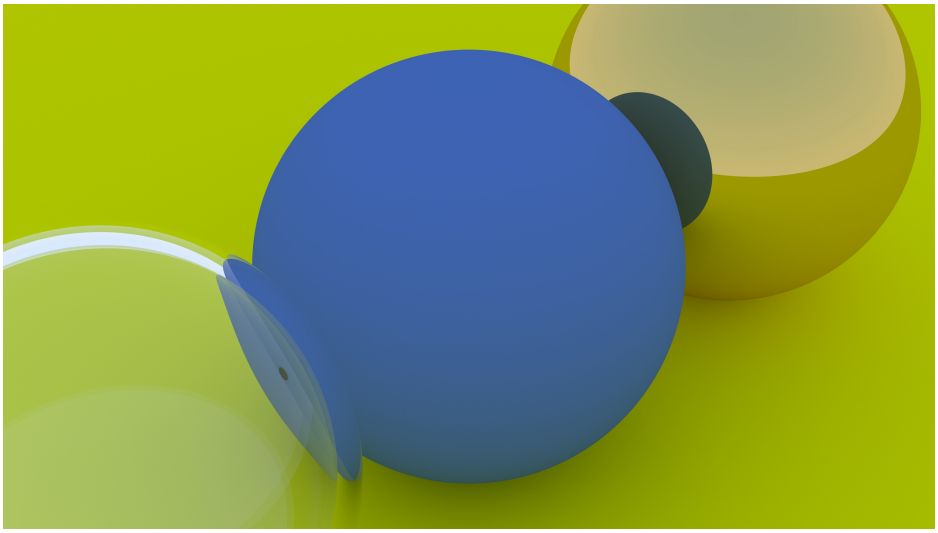

Figure 2: Hollow Glass Ball Small FOV

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

2.3 Hollow Glass Ball Off Focus

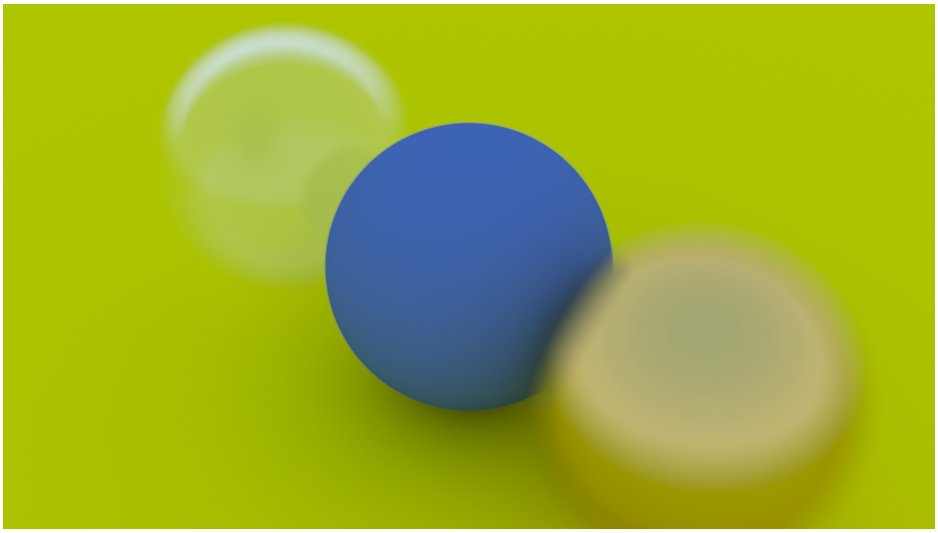

Figure 3: Hollow Glass Ball Off Focus

This scene shows off-focus blur effect.

2.4 Many Balls

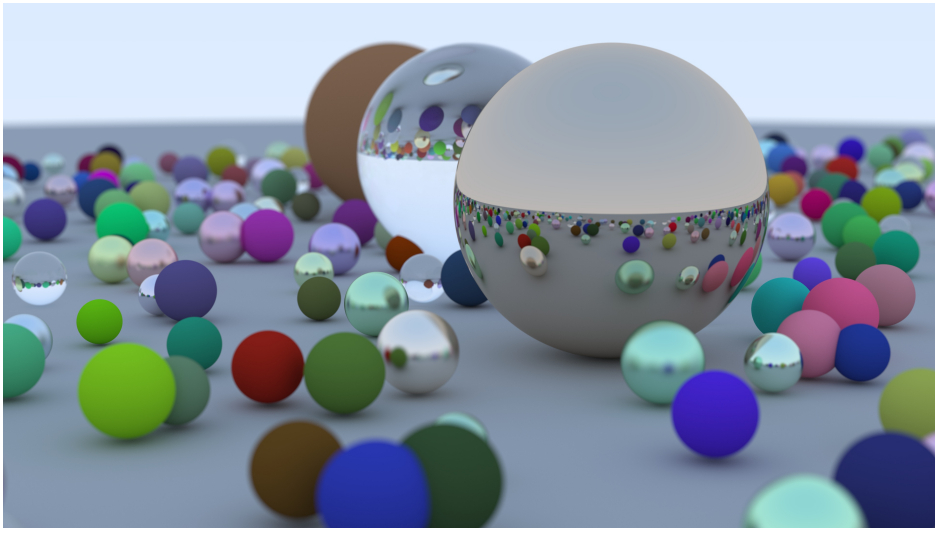

Figure 4: Many Balls

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

2.5 Motion Blur

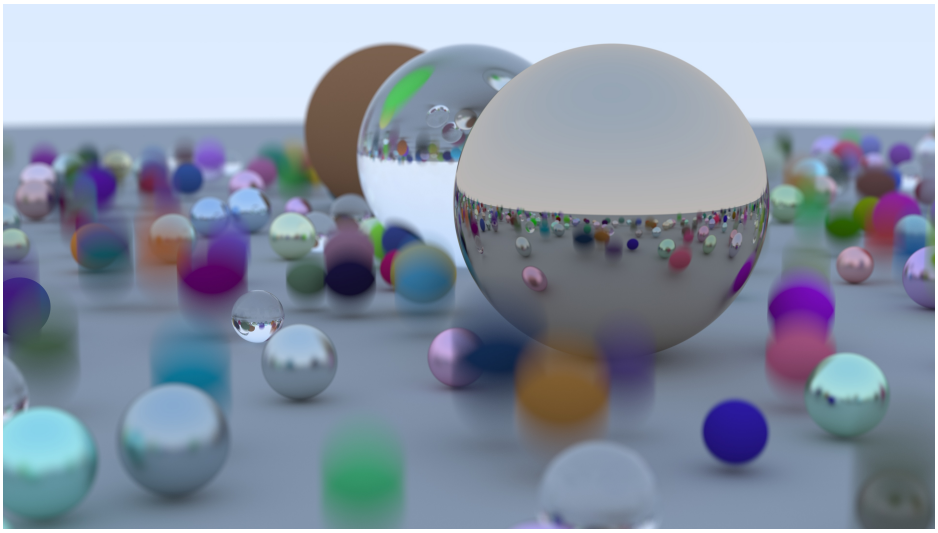

Figure 5: Motion Blur

This scene shows motion blur effect.

2.6 Motion Blur Checker

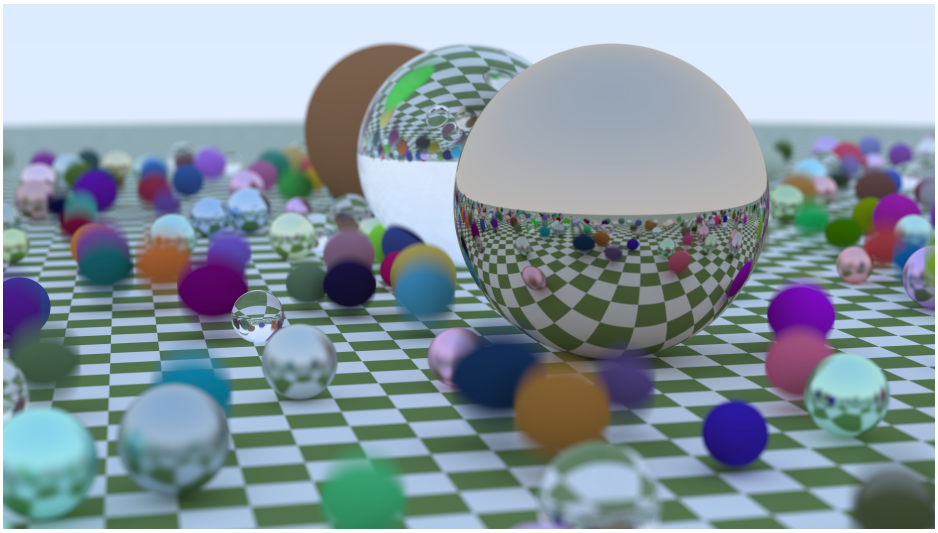

Figure 6: Motion Blur Checker

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

2.7 Earth

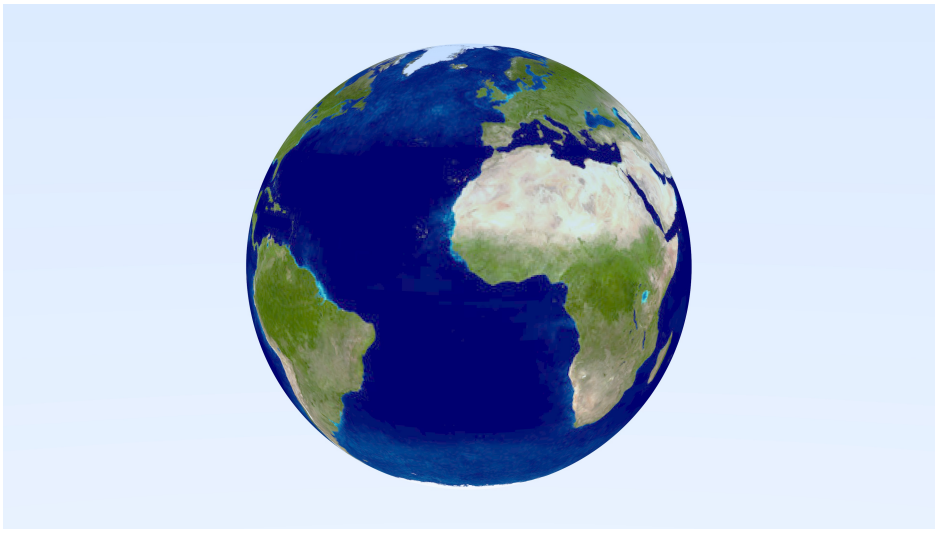

Figure 7: Earth

This scene is a ball with image texture.

2.8 Cornell Box Series

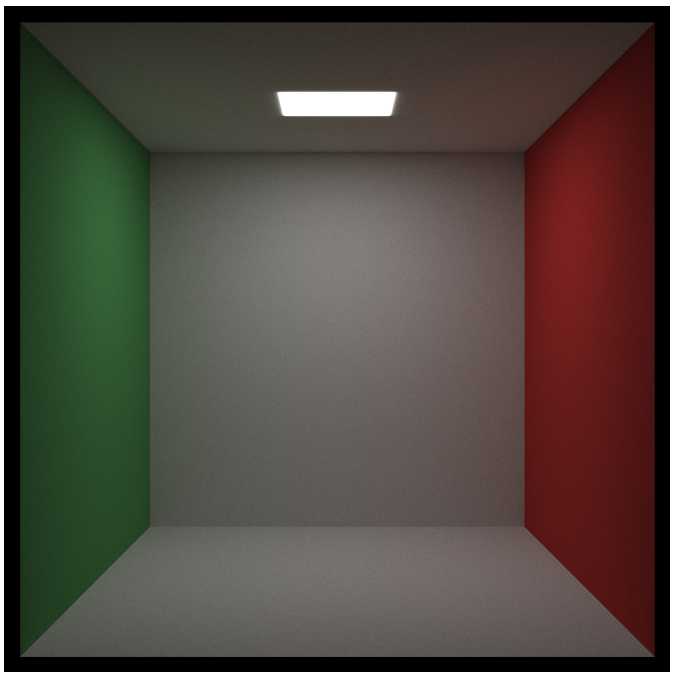

Figure 8: Cornell Box Empty

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

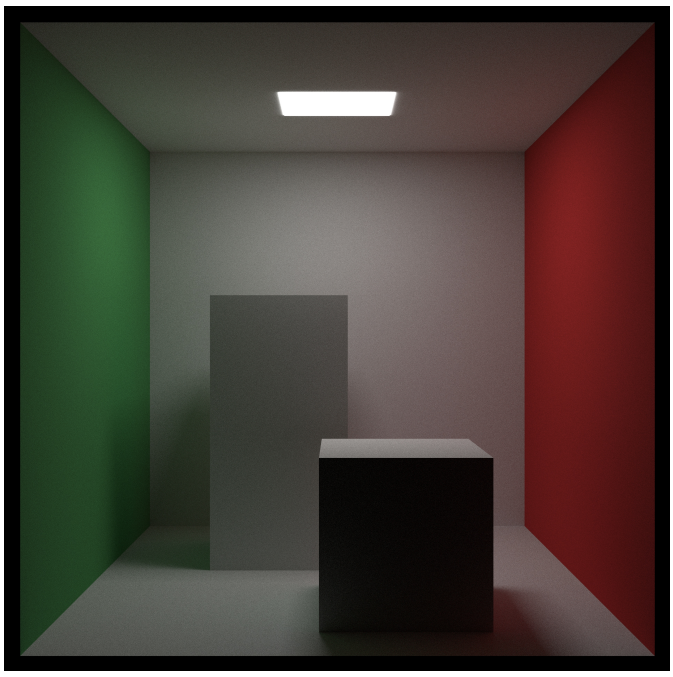

Figure 9: Cornell Box Tow Blocks

Added two boxes to the last scene.

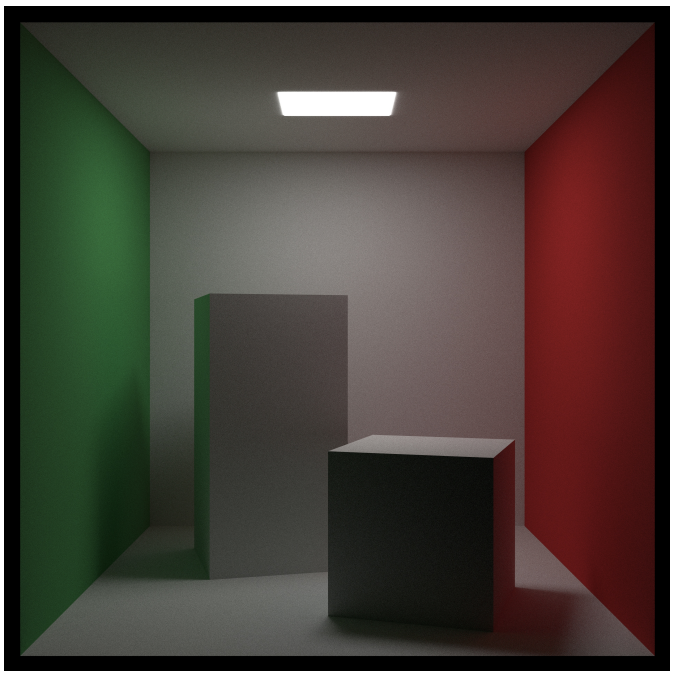

Figure 10: Cornell Box

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

2.9 Cornell Box Participating Media

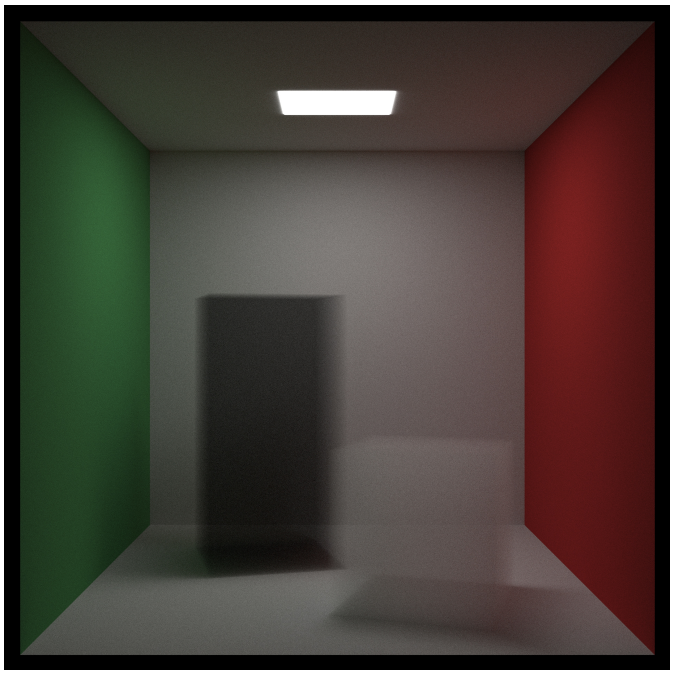

Figure 11: Cornell Box Participating Media

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

2.10 Book2 Final

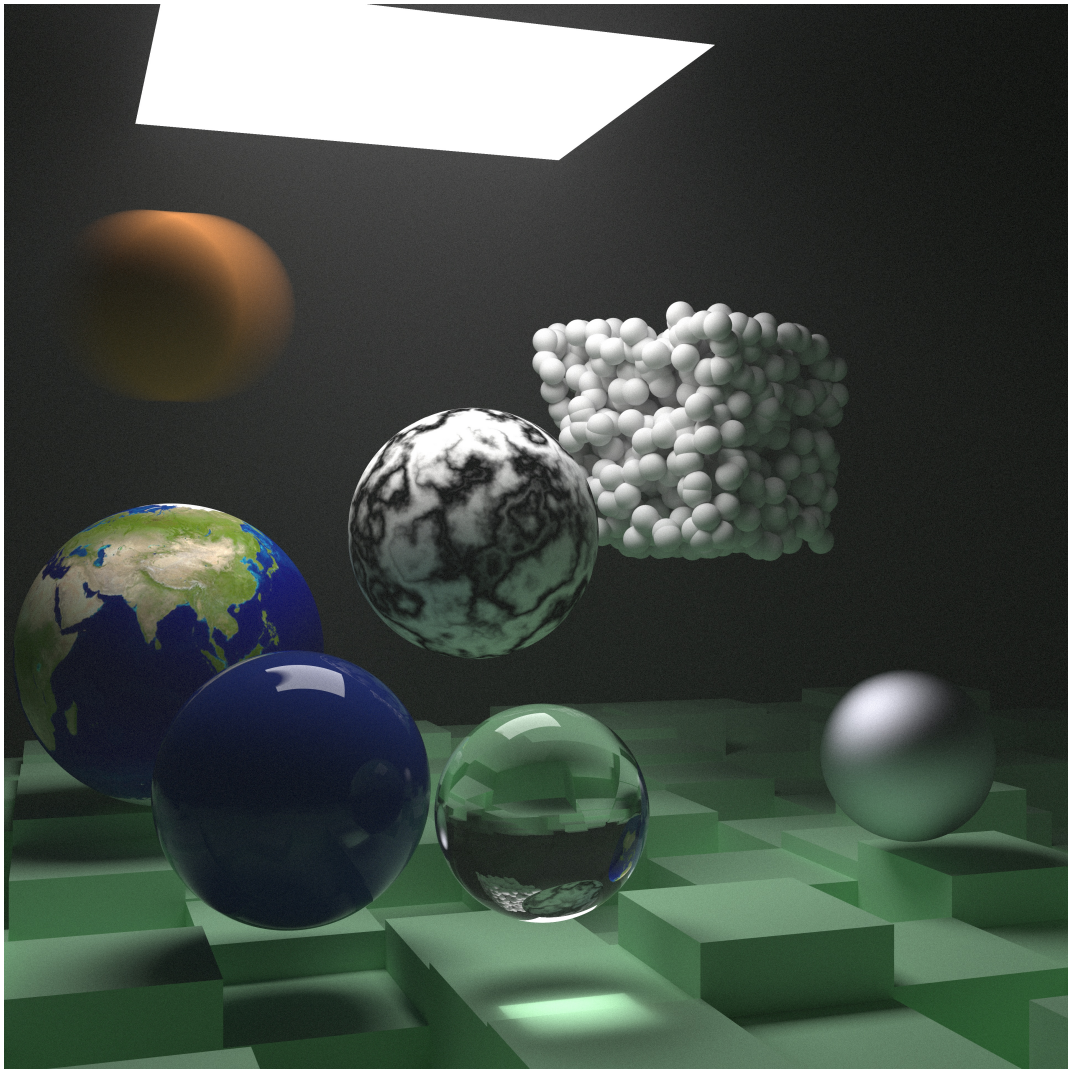

Figure 12: Book2 Final

This scene is a modified version of the one used by Peter Shirley in Ray Tracing Mini-Books 2. It combines may 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 use BVH to accelerate.

2.11 Test Obj

Figure 13: Test Obj

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

2.12 Sponza Sun

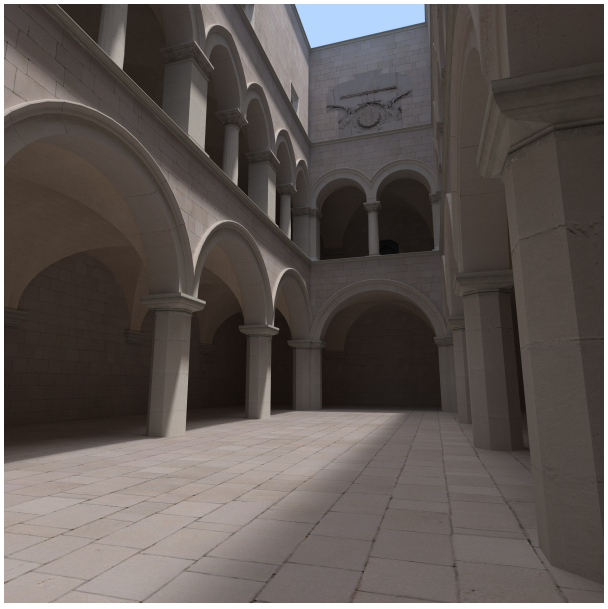

Figure 14: Sponza Sun

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

2.13 Sponza Crytek Series

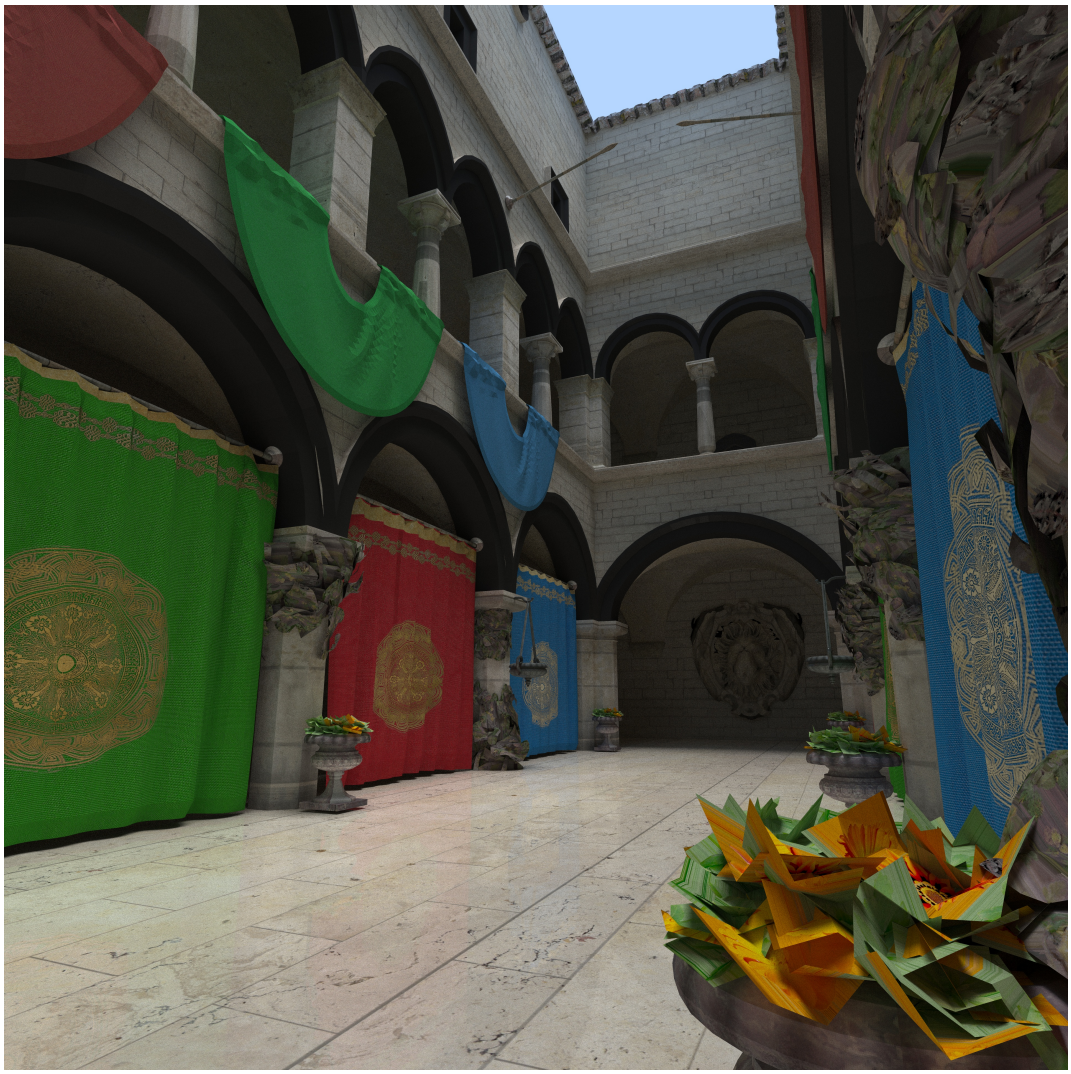

Figure 15: Sponza Crytek Cloudy

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

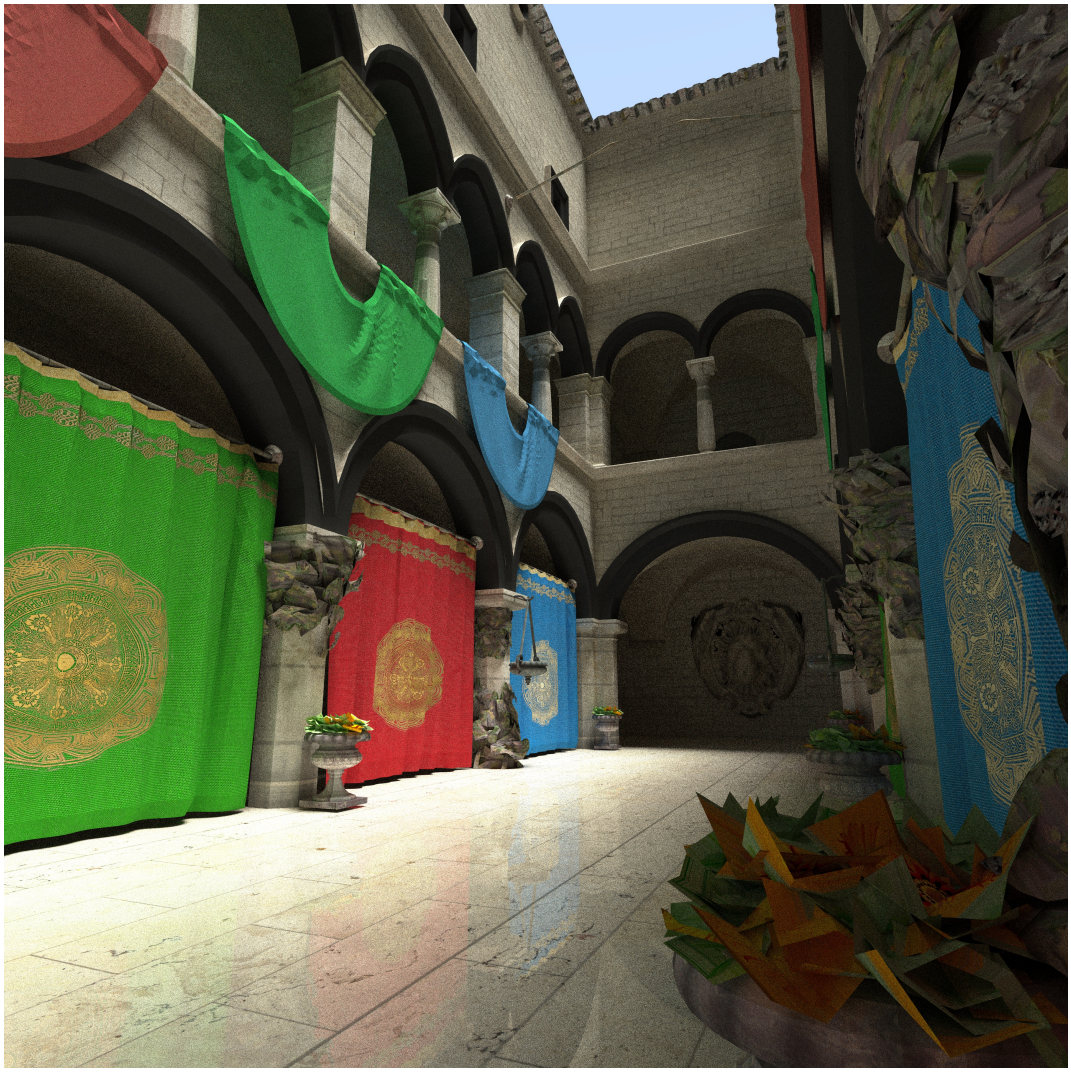

Figure 16: Sponza Crytek Sunny

2.14 Photon Mapping Series

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

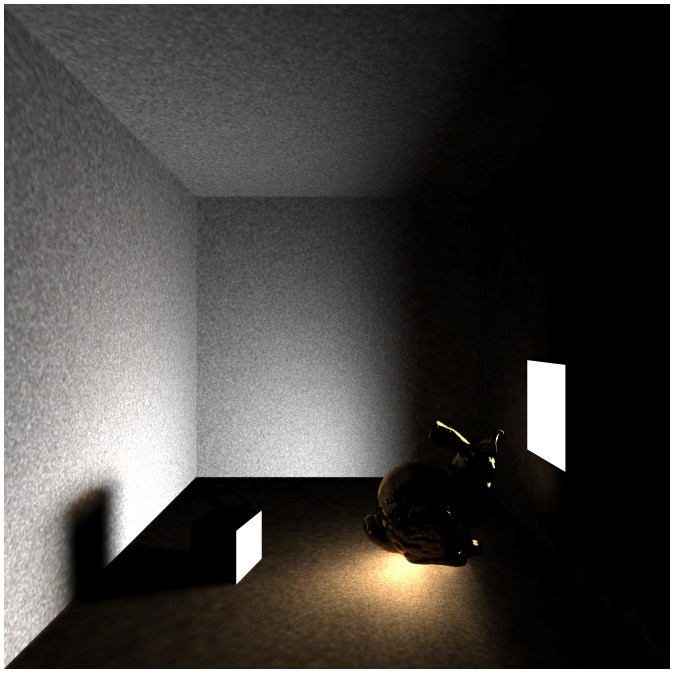

Figure 17: Photon Mapping Bunny Caustic

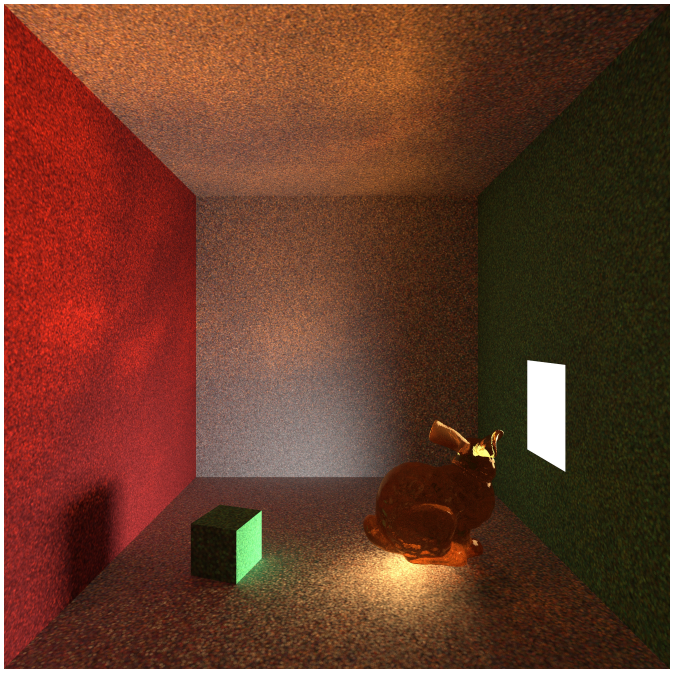

Figure 18: Photon Mapping Bunny

3 Usage

How set scene before compilation

step 1: set image settings (customization (e.g. changing aspect_ratio to 16.0 / 9.0) can be done by modifying render_scene in main.cpp)
step 2: choose integrator (Photon Mapping only supports one scene, Path Tracing has step 3 of setting scene)
step 3: set scene, camera and skybox (scene functions can be found in src/scenes)

How to run this renderer

Windows:
step 1: compile with CLion using MSVC compiler
step 2: run "Project.exe 8", here 8 means using 8 threads (after running we will get 8 .partial files)
step 3: run "packager.exe 8", here 8 means combining 8 partial files (after running we will get a .ppm image, which can be opened using OpenSeeIt)
step 4: run python convert.py, which converts .ppm into .jpg (note that opencv for python is required to run this)
Linux:

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

4 Renderer Function Menu

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

4.1 Core Renderer:

Integrator: Monte Carlo Path Tracing, Photon Mapping.
Accelerator: BVH (AABB with SAH), Kd-Tree.
Hardware Acceleration: OpenMP on Windows, Multi-processing on Linux.

4.2 Objects:

Hittable Objects: Triangle, Triangular Mesh, Box, Sphere.
Complex Models: .obj Model with .mtl Material Description.
Skybox: Constant Skybox, Directional Skybox, Realistic Skybox, Multi-layer Skybox.
Transforms: Rotation, Translation.

4.3 Materials & Textures

Materials: Lambertian, Metal, Dielectric, PBR material, Isotropic, Diffuse Light.
Textures: Color Texture, Checker Texture, Perlin Noise Texture, Marble Texture, Image Texture, Bump Texture.

4.4 Visual Effects

Camera Effects: Off-focus Blur, Motion Blur.
Volumetric Rendering: Participating Media.

Other Visual Effects: Gamma Correction, Caustic.

5 Renderer Infrastructures

5.1 Math Utilities

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

```cpp
//
// Created by meiyixuan on 2021-12-10.
//

#ifndef PROJECT_UTILS_H
#define PROJECT_UTILS_H

// path
inline std::string model_pth() {
  #if defined(WINDOWS)
   return "./";
  #else
   return "./resources/";
  #endif
}

// memory
using std::shared_ptr;
using std::make_shared;

// statistics
int normal_triangles = 0;
int u_degrade_triangles = 0;
int v_degrade_triangles = 0;
int unrecoverable_triangles = 0;

// integrator
int integrator_type;

int use_photon_map() { return 1; }

int use_path_tracing() { return 0; }

// math
using std::sqrt;
const double inf = std::numeric_limits<double>::infinity();
const double TMIN = 0.001;
const double EPSILON = 0.0000001;
const double pi = 3.1415926535897932385;

inline double deg2rad(double deg) {
  return deg * pi / 180.0;
}

inline double clamp(double x, double min, double max) {
  if (x < min) return min;
  if (x > max) return max;
  return x;
```

```
}

// random
inline double random_double_fixed() {
  // random number in [0, 1), with fixed initial seed
  // This function is thread-identical, and should be used in
  // perlin noise and random scene generation.
  return rand() / (RAND_MAX + 1.0);
}

inline double random_double() {
  if (integrator_type == 1) {
    return random_double_fixed();
  }
  // random number in [0, 1)
  static std::random_device rd;
  static std::uniform_real_distribution<double> distribution(0.0, 1.0);
  static std::mt19937 generator(rd());
  return distribution(generator);
}

inline double random_double(double min, double max) {
  // random real in [min,max).
  return min + (max - min) * random_double();
}

inline double random_double_fixed(double min, double max) {
  // random real in [min,max).
  return min + (max - min) * random_double_fixed();
}

inline int random_int_fixed(int min, int max) {
  // Returns a random integer in [min,max].
  return static_cast<int>(random_double_fixed(min, max + 1));
}

// input
int parse_int(const char *buf, size_t len) {
  int res = 0;
  for (size_t i = 0; i < len; ++i) {
    res *= 10;
    res += buf[i] - '0';
  }
  return res;
}

#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)

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

```
//

#ifndef PROJECT_VECTOR3D_H
#define PROJECT_VECTOR3D_H

class Vector3d {
  public:
  // constructors
  Vector3d() : val{0, 0, 0} {}

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

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

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

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

  // operators
  Vector3d &operator+=(const Vector3d &v) {
    val[0] += v.val[0];
    val[1] += v.val[1];
    val[2] += v.val[2];
    return *this;
  }

  Vector3d &operator-=(const Vector3d &v) {
    val[0] -= v.val[0];
    val[1] -= v.val[1];
    val[2] -= v.val[2];
    return *this;
  }

  Vector3d &operator*=(const double t) {
    val[0] *= t;
    val[1] *= t;
    val[2] *= t;
    return *this;
  }

  Vector3d &operator/=(const double t) {
    return *this *= 1 / t;
  }

  Vector3d operator-() const { return {-val[0], -val[1], -val[2]}; }

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

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

  double length() const {
    return std::sqrt(squared_length());
  }

  double squared_length() const {
    return val[0] * val[0] + val[1] * val[1] + val[2] * val[2];
```

```
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();
```

```
  if (dot(in_unit_sphere, normal) > 0.0)
  return in_unit_sphere;
  else
  return -in_unit_sphere;
}

Vector3d reflect(const Vector3d &v, const Vector3d &n) {
  return v - 2 * dot(v, n) * n;
}

Vector3d refract(const Vector3d &uv, const Vector3d &n, double refract_coefficient) {
  // eta: typically air = 1.0, glass = 1.3 - 1.7, diamond = 2.4
  // refract_coefficient is equal to eta_in / eta_out
  auto cos_theta = fmin(dot(-uv, n), 1.0);
  Vector3d r_out_perp = refract_coefficient * (uv + cos_theta * n);
  Vector3d r_out_parallel = -sqrt(fabs(1.0 - r_out_perp.squared_length())) * n;
  return r_out_perp + r_out_parallel;
}

#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)

```
//
// Created by meiyixuan on 2021-12-18.
// Perlin noise generator uses code from Peter Shirley.
// Warning: Perlin noise generator must have identical seeds across all processes.
//

#ifndef PROJECT_PERLIN_H
#define PROJECT_PERLIN_H

class PerlinNoise {
  public:
  PerlinNoise() {
    rand_vector = new Vector3d[point_count];
    for (int i = 0; i < point_count; ++i) {
      rand_vector[i] = normalize(Vector3d::random_fixed(-1, 1));
    }

    perm_x = perlin_generate_perm();
    perm_y = perlin_generate_perm();
    perm_z = perlin_generate_perm();
  }

  ~PerlinNoise() {
    delete[] rand_vector;
    delete[] perm_x;
    delete[] perm_y;
    delete[] perm_z;
  }

  double turbulence(const Point &p, int sample_depth = 7) const {
    // initialize
```

```
    auto sum = 0.0;
    auto cur_point = p;
    auto weight = 1.0;

    // weighted sum of samples
    for (int i = 0; i < sample_depth; i++) {
      sum += weight * noise(cur_point);
      weight *= 0.5;
      cur_point *= 2;
    }
    return fabs(sum);
  }

  double noise(const Point &p) const {
    // only one sample from perlin noise
    auto u = p.x() - floor(p.x());
    auto v = p.y() - floor(p.y());
    auto w = p.z() - floor(p.z());
    auto i = static_cast<int>(floor(p.x()));
    auto j = static_cast<int>(floor(p.y()));
    auto k = static_cast<int>(floor(p.z()));
    Vector3d c[2][2][2];

    for (int di = 0; di < 2; di++)
    for (int dj = 0; dj < 2; dj++)
    for (int dk = 0; dk < 2; dk++)
    c[di][dj][dk] = rand_vector[
    perm_x[(i + di) & 255] ^
    perm_y[(j + dj) & 255] ^
    perm_z[(k + dk) & 255]];

    // tri-linear interpolation
    return perlin_interpolation(c, u, v, w);
  }

  private:
  static const int point_count = 256;
  Vector3d *rand_vector;
  int *perm_x;
  int *perm_y;
  int *perm_z;

  static int *perlin_generate_perm() {
    auto p = new int[point_count];
    for (int i = 0; i < PerlinNoise::point_count; i++)
    p[i] = i;
    permute(p, point_count);
    return p;
  }

  static void permute(int *p, int n) {
    for (int i = n - 1; i > 0; i--) {
      int target = random_int_fixed(0, i);
      int tmp = p[i];
      p[i] = p[target];
      p[target] = tmp;
    }
```

```
  }

  static double perlin_interpolation(Vector3d c[2][2][2], double u, double v, double w) {
    // hermite smoothing to eliminate Mach Band
    auto uu = u * u * (3 - 2 * u);
    auto vv = v * v * (3 - 2 * v);
    auto ww = w * w * (3 - 2 * w);

    // interpolate
    auto accum = 0.0;
    for (int i = 0; i < 2; i++)
    for (int j = 0; j < 2; j++)
    for (int k = 0; k < 2; k++) {
      Vector3d weight_v(u - i, v - j, w - k);
      accum += (i * uu + (1 - i) * (1 - uu))
      * (j * vv + (1 - j) * (1 - vv))
      * (k * ww + (1 - k) * (1 - ww))
      * dot(c[i][j][k], weight_v);
    }
    return accum;
  }
};

#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

5.3 External Libraries

6 References

Code References:

[1] Ray Tracing Mini-books, by Peter Shirley (https://raytracing.github.io)

[2] Physically Based Rendering: From Theory to Implementation, by Matt Pharr, Wenzel Jakob, and Greg Humphreys (https://github.com/mmp/pbrt-v3)

[3] Dezeming Family (https://dezeming.top/)

External Libraries:

[1] Tinyobjloader (https://github.com/tinyobjloader/tinyobjloader)

[2] stb_image (https://github.com/nothings/stb)

Model Resources:

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