Ray Tracer

Ray Tracer **1**
- Introduction 3
- How to run this code 3
- Code Structure 4
  - Scene 5
  - Camera 5
  - Material 5
  - HittableList 5
  - Primitive 6
  - HitRecord 6
  - Ray 6
  - Interval 6
  - Vec, Point, Colour 6
  - raytracer::utils, raytracer::vec 6
- Ray Tracer Algorithm 7
- Primitives 8
  - Sphere 8
  - Box 8
  - 2D Primitives 8
  - Mesh 8
- Phong Shading 9
  - Phong Material 9
  - Light Sampling and Soft Shadow effect 10
  - PhongMirror Material 11
- The Pixel Sampling Trade-off 12
- Results 13
  - Phong Cornell Box 13
  - Phong Spheres 14
  - Phong Quads 15
  - Mixed Materials (Phong + Dielectric) 16
  - Phong Bunny 17

Author: Caio Valente (https://github.com/valentecaio/raytracing)

Date: 2024-05-04

Introduction

A Physics Based Renderer is a software that simulates the physical behaviour of light in a scene to create realistic images. The renderer is based on the ray tracing algorithm, which simulates the path of light rays as they interact with objects in the scene. The renderer calculates the colour of each pixel in the image by tracing rays from the camera through the scene and calculating the interactions with the objects in the scene. An enhanced version of the algorithm can recursively cast rays into the scene to calculate indirect lighting. This is called path tracing.

Ray tracing is computationally expensive, but easy to parallelize, as each pixel can be calculated independently. This means that ray tracing is well suited for modern multi-core CPUs and GPUs.

The objective of this project is to implement a simple CPU ray tracer in C++ that can render scenes with basic geometric shapes and materials.

The developed ray tracer is able to render scenes with spheres, boxes, 2D quadrangles (quads) and triangles, and simulate the interactions of light with different types of material using the Phong reflection model or path tracing materials such as diffuse, metal, and dielectric materials. It is able to render scenes with point lights in sphere shapes and area lights in quads or triangles shapes and can be configured to run in parallel using OpenMP API.

How to run this code

This project only uses the header-only library GLM, that is included in the lib/ directory.

The compilation is done with g++ and the build is handled by make, using the Makefile provided in the root directory.

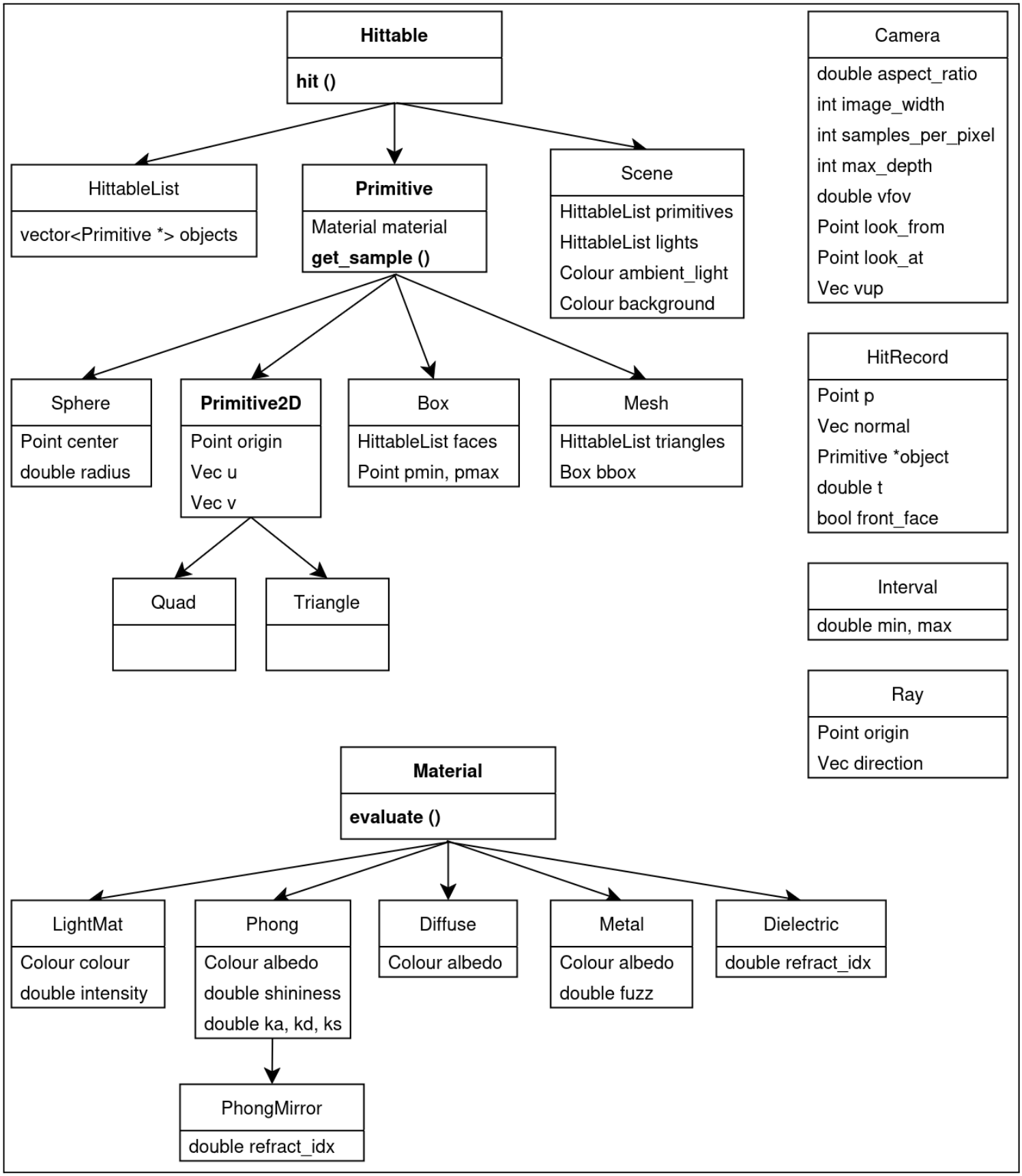
Enabling OpenMP to run the code in multi-threads can drastically reduce the time taken to render images. You can enable it when building the binary by defining OPENMP=1.

```
# compile code and save binary in build/raytracer
make

# or, in multi-threaded mode
make OPENMP=1

# generate image
build/raytracer > build/output.ppm
```

To build different scenes, change the scene number in the main() function at the end of the src/main.cpp file and recompile the code.

Code Structure

The ray tracer is implemented in C++ and consists of the classes presented in the following diagram.



Class Inheritance Relationships in the ray tracer (abstract classes and methods in bold)

Scene

Represents a hittable scene in 3D space. It contains a list of primitives and a list of lights that compose the geometric world that we want to render. Objects and lights are added to the scene using the add() method, which distinguishes between lights and other objects based on the material type. The scene also contains parameters such as the ambient light colour and the background colour, used as a default colour for when a ray misses. It derives from Hittable and implements the hit() method, which checks if any object or light in the scene was hit and returns its information.

Camera

Represents a camera that can render a scene. It contains most of the parameters of the algorithm, such as the aspect ratio, image width, samples per pixel, maximum depth of ray bounces, vertical field of view, camera location, camera target, and camera up vector. The samples_per_pixel parameter defines how many rays are sampled for each pixel, hugelly affecting the performance of the algorithm and the quality of the final results.

The camera also provides the entry point of the ray tracing algorithm - the render() method, which renders the image row by row, from top to bottom, by tracing rays through the scene and calculating the colour of each pixel. This method is actually a recursive path tracing algorithm, but it works as a simple ray tracing when there are no path tracing materials in the scene (i.e. only Phong, PhongMirror and LightMat).

Material

Represents a material that can be applied to objects in the scene. It is an abstract class that defines the interface for materials, that is the evaluate() method, which calculates the colour of a ray intersection point and the next ray to be traced (for path tracing materials).

The Phong and PhongMirror materials are implemented as subclasses of Material, which implement the evaluate() method to calculate the Phong shading and Phong shading with reflection using Schlick's approximation, respectively. They are meant to work with the ray tracing algorithm and so do not return a ray to be traced next.

Other materials such as Diffuse, Metal, and Dielectric are also implemented as subclasses of Material, but they are meant to work with the path tracing algorithm. They can be mixed with the Phong materials in the same scene without breaking the algorithm, but the results may not be accurate. These materials return a ray to be traced next by the path tracing algorithm.

The LightMat material is a special material that represents a light source in the scene. It can be added to any available Primitive object so it will be treated as a light source, but the results are not guaranteed to be accurate for all of them. When added to a Sphere, the object is treated as a point light source. When added to a Quad or Triangle, the object is treated as an area light source. When added to other primitives, they are also treated as area light sources, but the results are not guaranteed to be accurate.

HittableList

Represents a list of hittable geometry objects. The hit() method iterates over the list of objects and checks if the ray intersects each object, returning the closest intersection point.

Primitive

Represents a geometric primitive in the scene. It is an abstract class that defines the interface for primitives, that is the hit() method, which checks if a ray intersects the primitive and returns the hit record, and the get_sample() method, which returns a random sample point on the primitive (for area lights). The Sphere, Box, Quad, Triangle and Mesh classes are implemented as subclasses of Primitive, but only Quads and Triangles have proper implementations of get_sample() as of now, which makes them the only possible choices for area lights.

The Mesh primitive represents a mesh object loaded from an OBJ file. It makes use of other primitives to represent the Triangles that compose the mesh and the Box that bounds the mesh. It is a very simple and slow implementation and does not support textures. The bounding box is used to discard some of the rays that do not intersect the mesh, but it is not a BVH tree. There is a bunny file in assets/bunny.obj that can be used for tests.

All the primitive classes are described in the primitives/ directory.

HitRecord

Represents the information about a ray intersection point. It contains the intersection point, the normal at the intersection point, the material of the object that was hit, the t value of the ray at the intersection point, and the object that was hit.

Ray

Represents a ray in 3D space. It contains an origin point and a direction vector.

Interval

Represents an interval of real numbers. It is used to define the range of t values for ray intersections.

Vec, Point, Colour

The only classes not present in the diagram are Vec, Point and Colour, which are alias to the glm::dvec3 class. They represent a 3D vector, a 3D point and a colour in RGB space, respectively, in double precision. Since they are aliases to the same GLM class, they can be mixed in operations with GLM vectors and matrices.

raytracer::utils, raytracer::vec

The code also contains two utility namespaces: raytracer::utils and raytracer::vec. The first contains utility functions for random number generation and sampling, writing images, measuring time, and other general-purpose functions. The vec namespace contains complementary utility functions for vector operations that are not present in the GLM library. These utilities can be found in the utils/ directory.

Ray Tracer Algorithm

```
1    // returns the colour of the ray after it hits the scene
2    Colour trace_ray(const Ray& r_in, int depth) const {
3      // if we've exceeded the ray bounce limit, the ray was absorbed
4      if (depth <= 0)
5         return Colour(0,0,0);
6
7      // try to hit an object in the scene, starting at 0.0001 to avoid self-intersection
8      // misses are considered as background colour
9      HitRecord hit;
10     if (!scene.hit(r_in, Interval(0.0001, infinity), hit))
11       return scene.background;
12
13     // HIT //
14
15     Colour c;
16     Ray r_out;
17     if (hit.object->material->evaluate(scene, r_in, hit, c, r_out))
18       return c * trace_ray(r_out, depth-1); // ray bounced
19     return c; // ray was absorbed
20   }
```

camera.hpp

The render() method of the camera class is the entry point of the ray tracing algorithm. It uses the trace_ray() method to calculate the colour of each pixel. For each pixel in the film, it casts multiple rays (samples_per_pixel) and averages the colours to reduce noise.

The trace_ray() method contains the main logic of the ray tracer and path tracer algorithms.

The parameter r_in is a ray whose origin is at the camera centre, and direction is the normalised vector from the camera centre to a pixel in the image plane. The method returns the colour of the pixel in the image plane.

It works by casting a ray into the scene and checking if it intersects any object, including primitives and lights. If the ray intersects an object, it calls the evaluate() method of the material of the object to calculate the colour of the intersection point. If the material is a LightMat, it returns the radiance of the light source. If the material is a Phong or PhongMirror, it calculates the shading using the Phong reflection model. If the material is a Diffuse, Metal, or Dielectric, it calculates the shading using the path tracing algorithm, i.e., the ray "bounces" and the method returns a new ray (r_out) to be traced next. The colour of the intersection point is calculated recursively by calling trace_ray() with the new ray.

There is also a recursive limit (max_depth) to avoid infinite recursion in the path tracing algorithm. If the recursion limit is reached, the method returns a black colour, meaning that the ray did not hit any light source and was absorbed by the material. This limit is not important if only Phong materials are used in the scene, as they do not trigger the recursion.

Primitives

Sphere

The Sphere primitive is a simple geometric object that represents a sphere in 3D space. It is defined by its center and radius.

The hit() method solves the quadratic equation for the ray-sphere intersection, and returns the nearest intersection point within the acceptable range. The normal is normalised by the radius.

The get_sample() method returns the center of the sphere, which is why sphere lights are point lights.

Box

The Box primitive is a simple 3D box composed of a list of 6 quads. The constructor receives two Points a and b that define the opposite corners of the box and then constructs the 6 quads that compose the box.

The hit() method checks if the ray intersects any of the 6 quads and sets the object pointer to the box.

The get_sample() method returns a random point in one of the 6 box faces.

2D Primitives

The Primitive2D class is an abstract class that represents an instance of a 2D geometric object in the scene. These primitives are defined by an origin point and two vectors that define the plane where they lie. Each derived class must implement the is_hit() method that checks if a point with planar coordinates (alpha, beta) is inside the primitive boundaries. There are two derived classes: Quad and Triangle.

A Quad checks if the point is inside the boundaries by checking if the planar coordinates are between 0 and 1.

A Triangle checks if the point is inside the boundaries by checking if the sum of the planar coordinates is less than or equal to 1.

The get_sample() of each primitive returns a random point inside the primitive boundaries.

Mesh

The Mesh primitive is a list of triangles with a bounding box to speed up the intersection tests. This is still a simple implementation, very inefficient and experimental. The constructor can create a mesh from a list of triangles or from an obj file. To simplify code, it makes use of the existing Triangle and Box primitives, although it is not the most efficient way to implement a mesh.

The hit() method checks if the ray intersects the bounding box first and then checks if the ray intersects any of the triangles.

The get_sample() method returns a random point in one of the triangles.

The load_obj() method loads a mesh from an obj file (only simple triangulated meshes are supported).

The compute_bbox() method is called by the constructor and computes the bounding box of the mesh.

Phong Shading

Phong Material

```
bool evaluate(const Scene& scene, const Ray& r_in, const HitRecord& hit,
              Colour& out_colour, Ray& out_ray) const override {
  out_colour = phong_shade(r_in, hit, scene);
  return false;
}

Colour phong_shade(const Ray& r_in, const HitRecord& hit, const Scene& scene) const {
  Colour total_amb = scene.ambient_light;
  Colour total_diff = Colour(0,0,0);
  Colour total_spec = Colour(0,0,0);

  // view direction is the opposite of the incoming ray direction (already normalized)
  Vec view_dir = -r_in.direction();

  // try to hit lights in the scene to calculate the shading
  for (const auto& light : scene.lights.objects) {
    HitRecord shadow_hit;
    auto diff = Colour(0,0,0);
    auto spec = Colour(0,0,0);

    // point lights are sampled once, area lights are sampled multiple times
    int nsamples = (std::dynamic_pointer_cast<Sphere>(light)) ? 1 : 10;
    for (int i = 0; i < nsamples; i++) {
      Point sample = light->get_sample();
      Vec light_dir = glm::normalize(sample - hit.p);

      auto shadow_ray = Ray(hit.p, light_dir);
      if (scene.hit(shadow_ray, Interval(0.0001, infinity), shadow_hit)
          && shadow_hit.object == light) {
        // light is visible from the hit point
        auto light_mat = std::dynamic_pointer_cast<LightMat>(light->material);

        // diffuse
        Vec light_radiance = light_mat->radiance(glm::length(sample - hit.p));
        double attenuation = std::max(glm::dot(hit.normal, light_dir), 0.0);
        diff += attenuation * light_radiance;

        // specular
        Vec reflect_dir = glm::normalize(glm::reflect(-light_dir, hit.normal));
        double RdotV = std::pow(std::max(glm::dot(reflect_dir, view_dir), 0.0), shininess);
        spec += light_radiance * RdotV;
      }
    }
    total_diff += diff/(double)nsamples;
    total_spec += spec/(double)nsamples;
  }
  return albedo * (ka*total_amb + kd*total_diff + ks*total_spec);
}
```

The Phong material uses the Phong reflection model to calculate the shading of the object. It combines ambient, diffuse and specular lighting to calculate the final colour of the object. Each instance of the material has a different albedo, shininess, ambient, diffuse and specular coefficients.

The albedo represents the colour that the material reflects. The shininess is the exponent of the specular term in the Phong model and determines the size of the specular highlight.

The Phong shading is made by calculating the ambient, diffuse and specular components of the Phong model for each light in the scene. The ambient component represents the light that is reflected by the object without any direct light source. The diffuse component represents the light that is reflected by the object in all directions. The specular component represents the light that is reflected by the object in a mirror-like way.

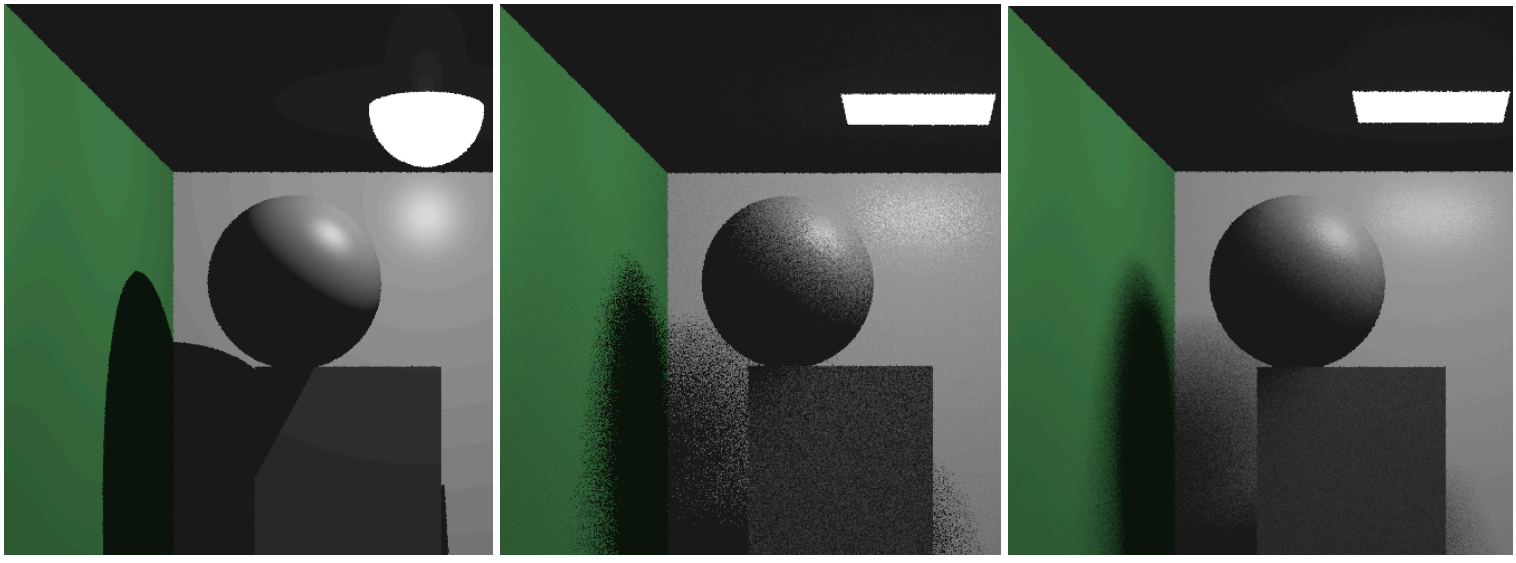
To calculate the diffuse component, a ray is traced towards each light component. If it hits the light without any obstruction, this light contributes to the diffuse component of this material.

The final colour is calculated as the sum of the ambient, diffuse and specular components of the Phong model, weighted by the ambient, diffuse and specular coefficients, and multiplied by the albedo of the material.

Light Sampling and Soft Shadow effect

When tracing rays towards lights in the calculation of the diffuse component of Phong shading, the algorithm needs to decide where exactly in the surface of the light it should point these rays to. Different choices will lead to different results. A common approach consists in choosing a predetermined number of samples randomly distributed in the surface of the light shape. This should create a soft shadow effect. To correctly implement this feature, the algorithm should be able to generate uniformly distributed random samples in the area of the light surface. As of now, only 2D primitives can properly achieve such, which means that they are the only options for area lights.

When a sphere is added as a light source, it is considered as a point light, and the algorithm will sample it once, at the center of the sphere. Area light sources are sampled 10 times.



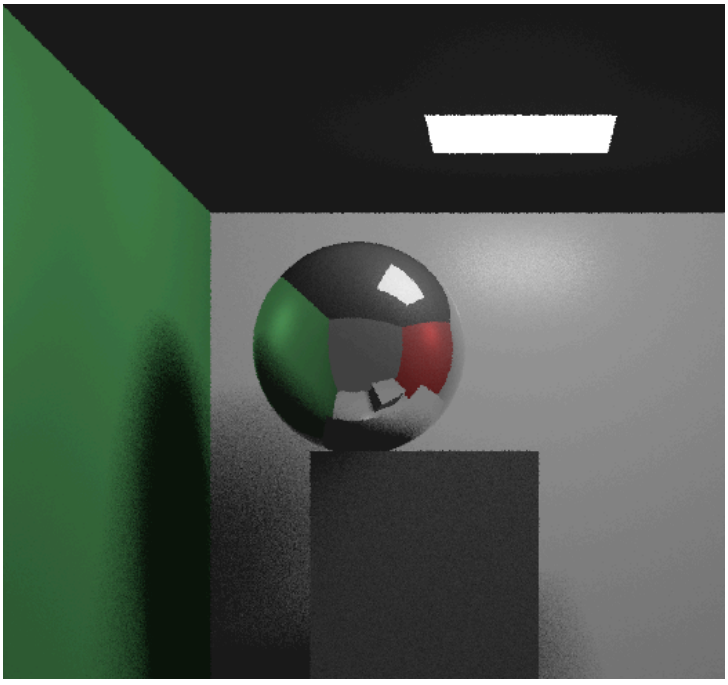
(A) (B) (C)

Side by side comparison of different light sampling:
(A) A sphere point light sampled 1 time;
(B) A quad area light sampled 1 time;
(C) A quad area light sampled 10 times;

PhongMirror Material

```
bool evaluate(const Scene& scene, const Ray& r_in, const HitRecord& hit,
              Colour& out_colour, Ray& out_ray) const override {
  // reflection ray
  Vec reflected = glm::normalize(glm::reflect(r_in.direction(), hit.normal));
  Ray reflect_ray(hit.p, reflected);

  // evaluate the reflected ray colour
  HitRecord reflect_hit;
  Colour reflect_colour;
  if (scene.hit(reflect_ray, Interval(0.0001, infinity), reflect_hit)) {
    // hit, evaluate the material
    reflect_hit.object->material->evaluate(scene, reflect_ray, reflect_hit,
                                           reflect_colour, out_ray);
  } else {
    // miss, use scene background colour
    reflect_colour = scene.background;
  }

  // reflectance
  double cos_theta = std::min(glm::dot(-r_in.direction(), hit.normal), 1.0);
  double R = utils::reflectance(cos_theta, refract_idx);

  // final colour is a mix of the phong shading and reflected colour
  out_colour = (1-R)*phong_shade(r_in, hit, scene) + R*reflect_colour;
  return false;
}
```

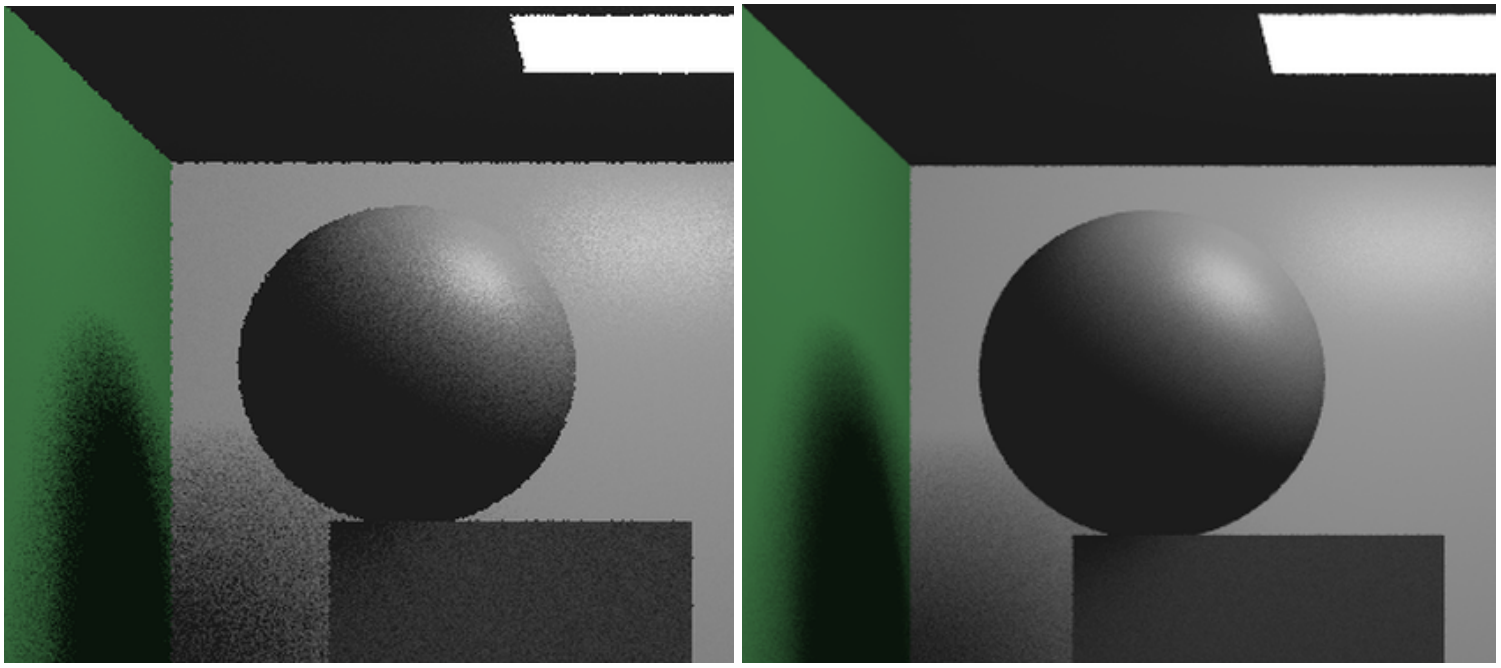
The PhongMirror material extends the Phong material by adding reflection to the pixel evaluation. It uses Schlick's approximation to calculate the reflectance of the material, which is a simple approximation of the Fresnel equations. The material has a refractive index that determines how much light is reflected and how much is refracted. A reflection ray is cast into the scene and evaluated. The final colour is a mix of the Phong shading and the colour of the reflected ray, weighted by the calculated reflectance of the material, resulting in a mirror-like material.



The Pixel Sampling Trade-off

The camera object has a parameter called samples_per_pixel that defines how many rays are sampled for each pixel in the viewport matrix. This is used to reduce the aliasing effect in the final image. It is a direct trade-off between quality and rendering time.

A pixel is a point, but in the context of computer graphics, it is a square area in the image plane. By uniformly sampling rays in the pixel area, we can estimate the final colour of the whole area by averaging the colours of the rays that hit the scene.



(A) (B)

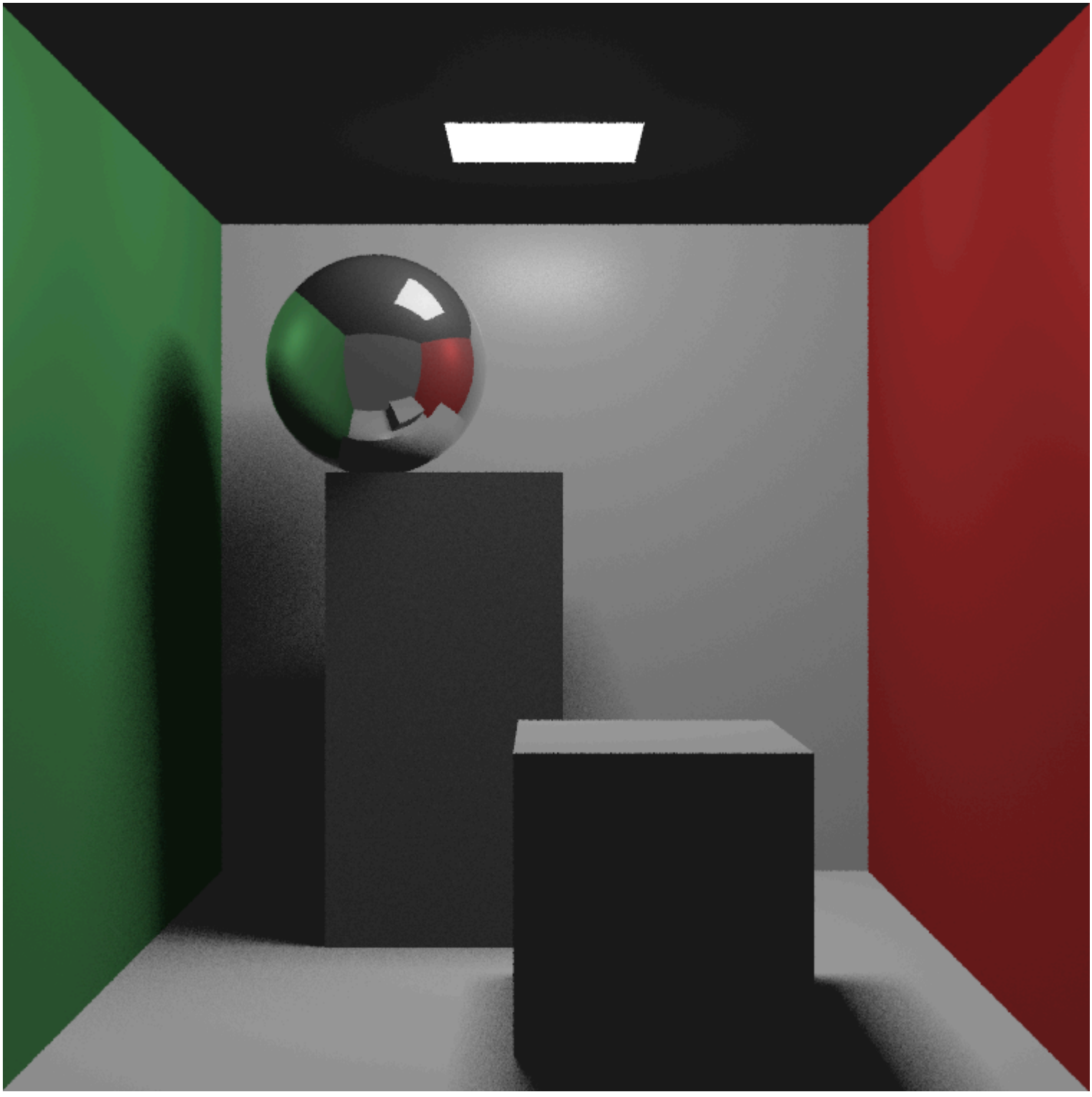
Side by side comparison of different pixel sampling:
(A) 1 sample per pixel (2.9 seconds);
(B) 10 samples per pixel (28.2 seconds);

Results

Phong Cornell Box

This version of the classic Cornell Box was composed with Phong and PhongMirror materials, using quads, boxes, a sphere and a quad light. The walls were made of quad primitives, and a reflective sphere was added. The background colour is grey, hence the grey wall being reflected in the center of the sphere.

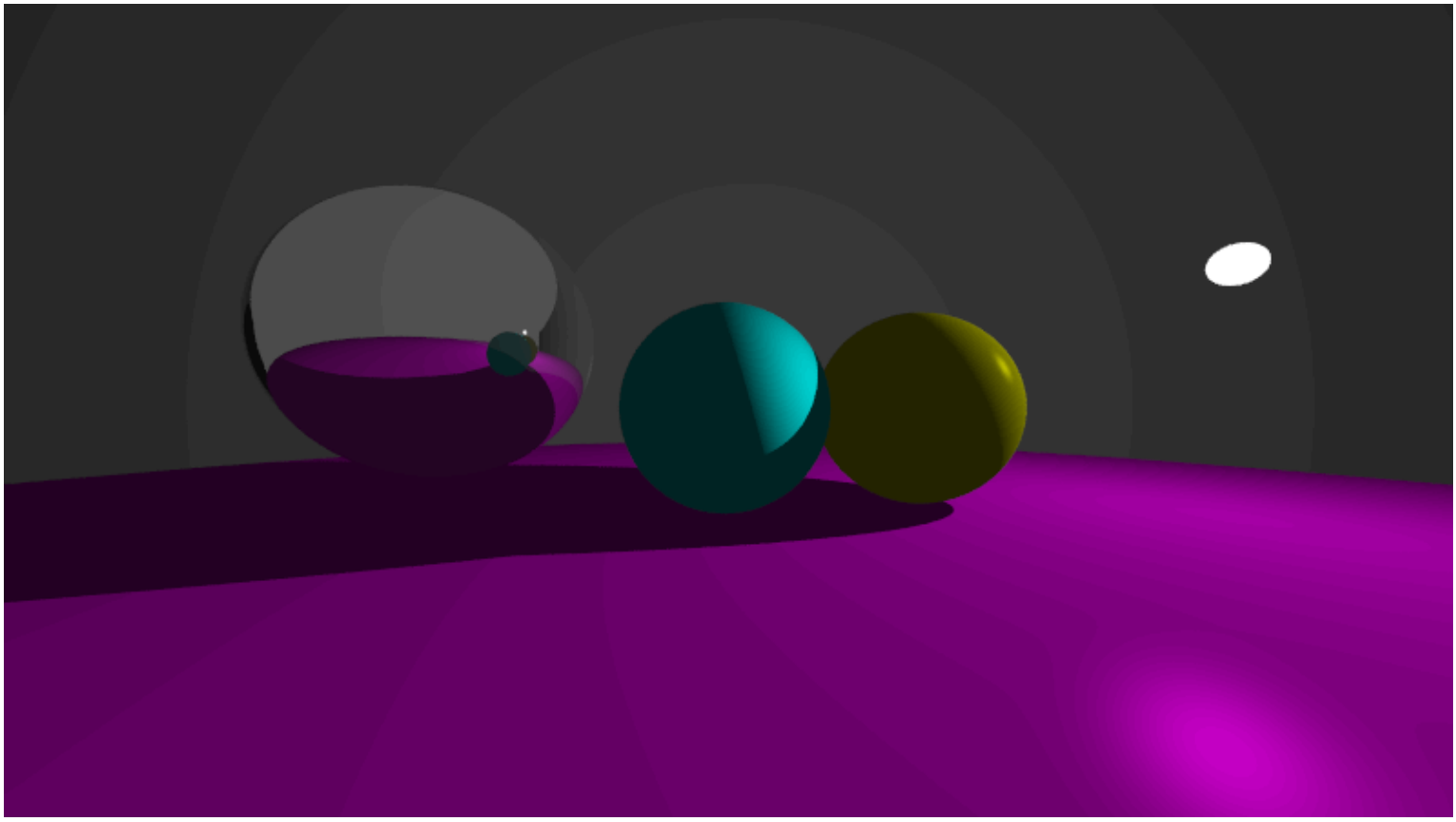
The following image was rendered using 5 rays per pixel in 800x800 pixels. The Phong materials trace 10 rays per light source. It took around 14 seconds to be generated in multi-thread (Intel i5-12450H, 12 cores) or around 41 seconds in single-core (800x800 pixels, 5 samples per pixel).



Phong Spheres

This scene contains spheres and a point light. It is useful to compare different shininess coefficients. The pink (ground) and yellow spheres have a shininess of 100, while the cyan sphere has a value of 2. We can see that a higher coefficient results in a more specular reflection, while a lower coefficient results in a more diffuse reflection. The mirror sphere has a PhongMirror material with a shininess of 1000 and a refraction index of 0.02.

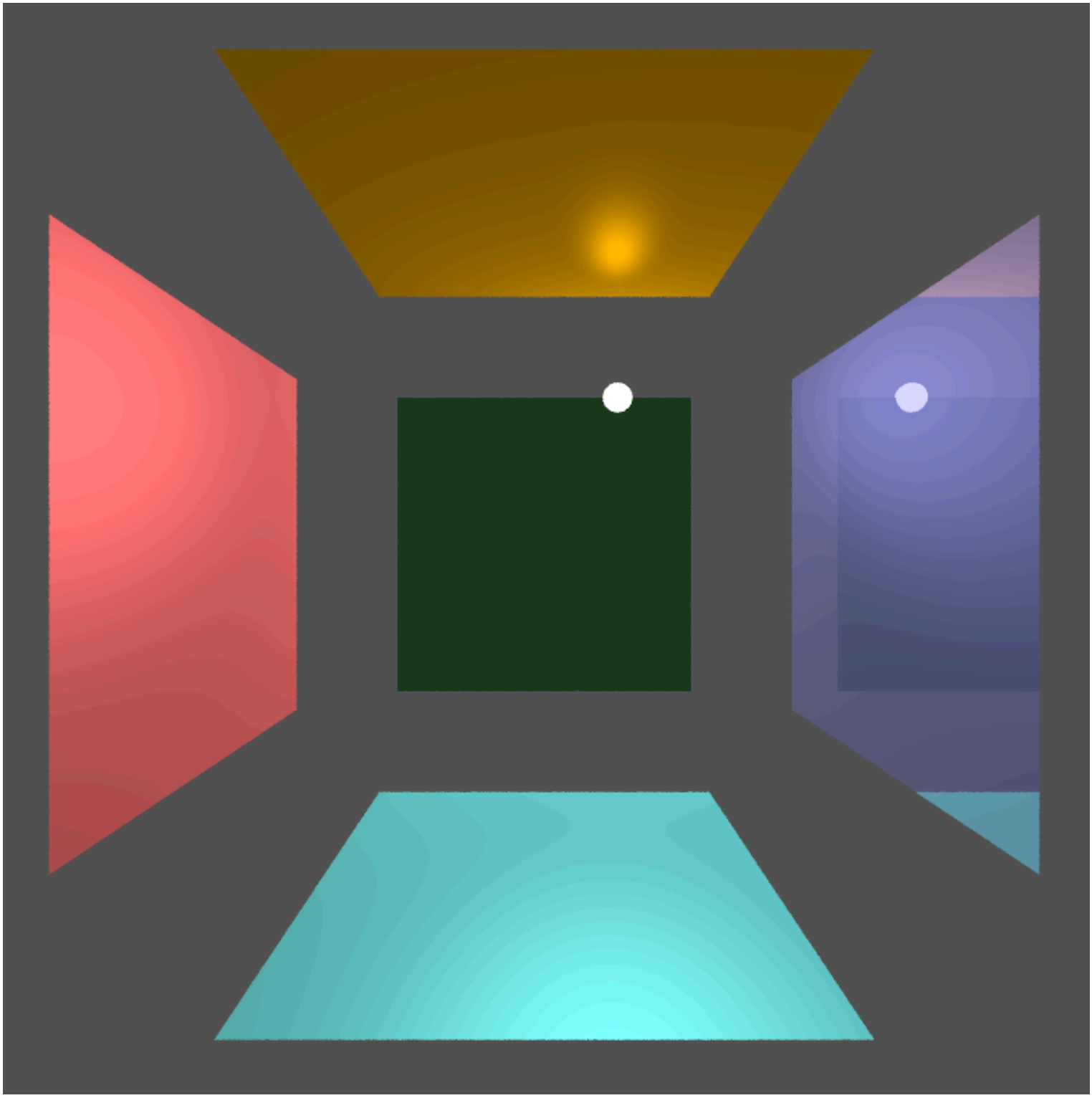
The scene is rendered with a camera resolution of 800x450 pixels (16/9 aspect ratio), 20 rays per pixel, and takes around 3.7 seconds using 12 threads or 9.2 seconds in single-thread.



Phong Quads

This scene contains quads with Phong materials and a point light. The quads have different shininess coefficients. The right quad is a PhongMirror material with a shininess of 10 and a reflectivity of 0.1. The scene has a strong ambient light, which makes the ambient term of the Phong reflection model more visible, making it easier to see the effect of the shininess coefficient on the mirror-like reflection.

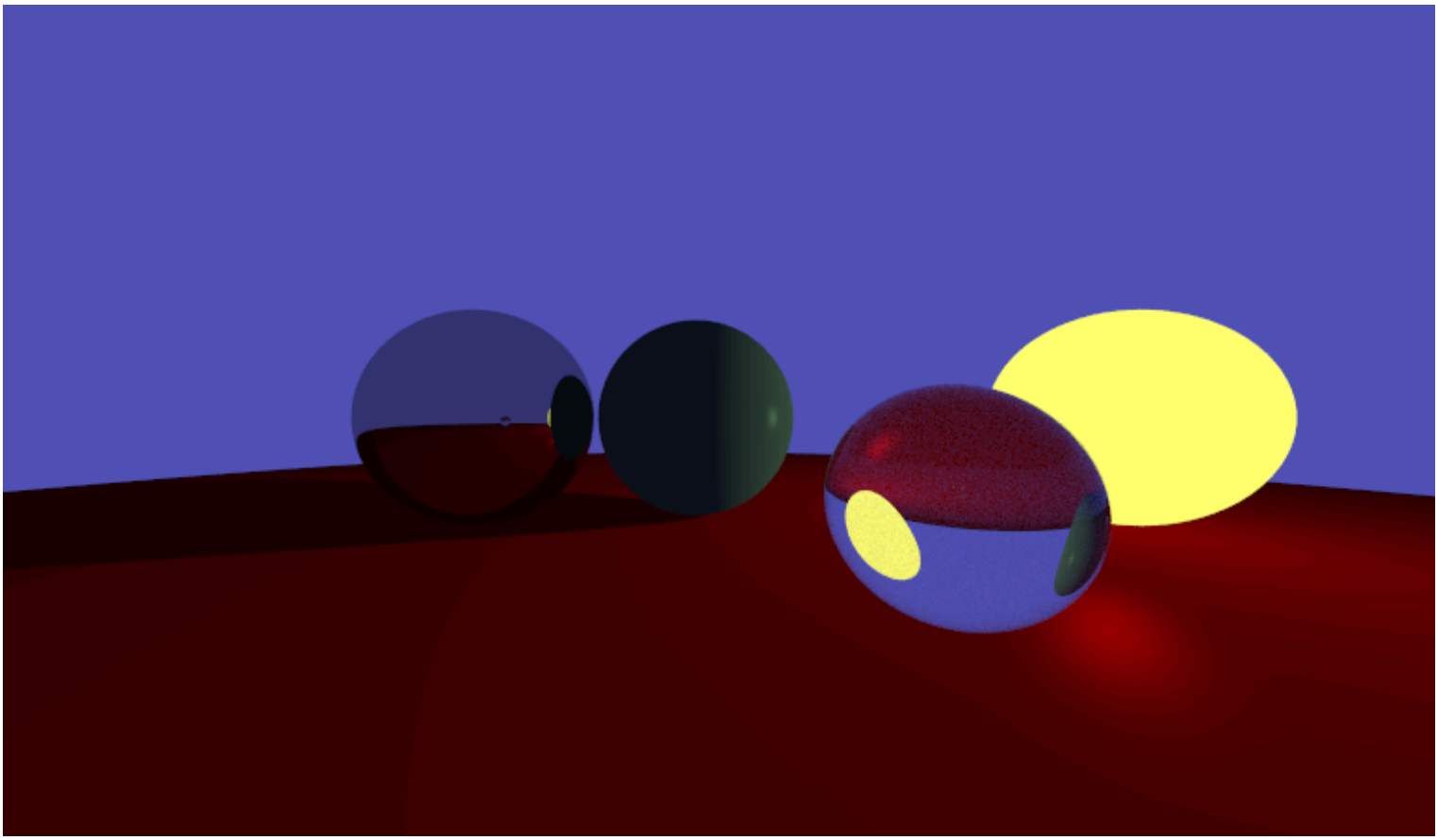
The scene is rendered with a resolution of 800x800 pixels, 10 rays per pixel, in around 2.7 seconds (12 cores) or 5.5 seconds single-core.



Mixed Materials (Phong + Dielectric)

This scene mixes Phong spheres with a Dielectric path traced sphere. The yellow sphere is a point light. We can see that the path traced sphere has some noise in it.

This image was generated at 800x450 resolution, using 50 rays per pixel, as path traced materials need more samples per pixel to reduce noise. It averages 8.4 seconds to be generated using 12 threads.



Phong Bunny

This scene contains a Phong Mesh of a bunny, a PhongMirror sphere, and a point light. The bunny is loaded from assets/bunny.obj and has 4968 triangles.

Adding a Mesh to the scene substantially increases the rendering time, as the ray tracer has to check for intersections with each triangle of the mesh every time the bounding box of the mesh is hit.

The image below was rendered with a resolution of 800x450 pixels, 10 rays per pixel, in 68 seconds using 12 cores (i5-12450H). The single-core test took 257 seconds. A bit far from real-time.

The time could be improved by implementing a BVH tree to speed up the intersection tests with the mesh triangles or by using better bounding box and triangle representations.