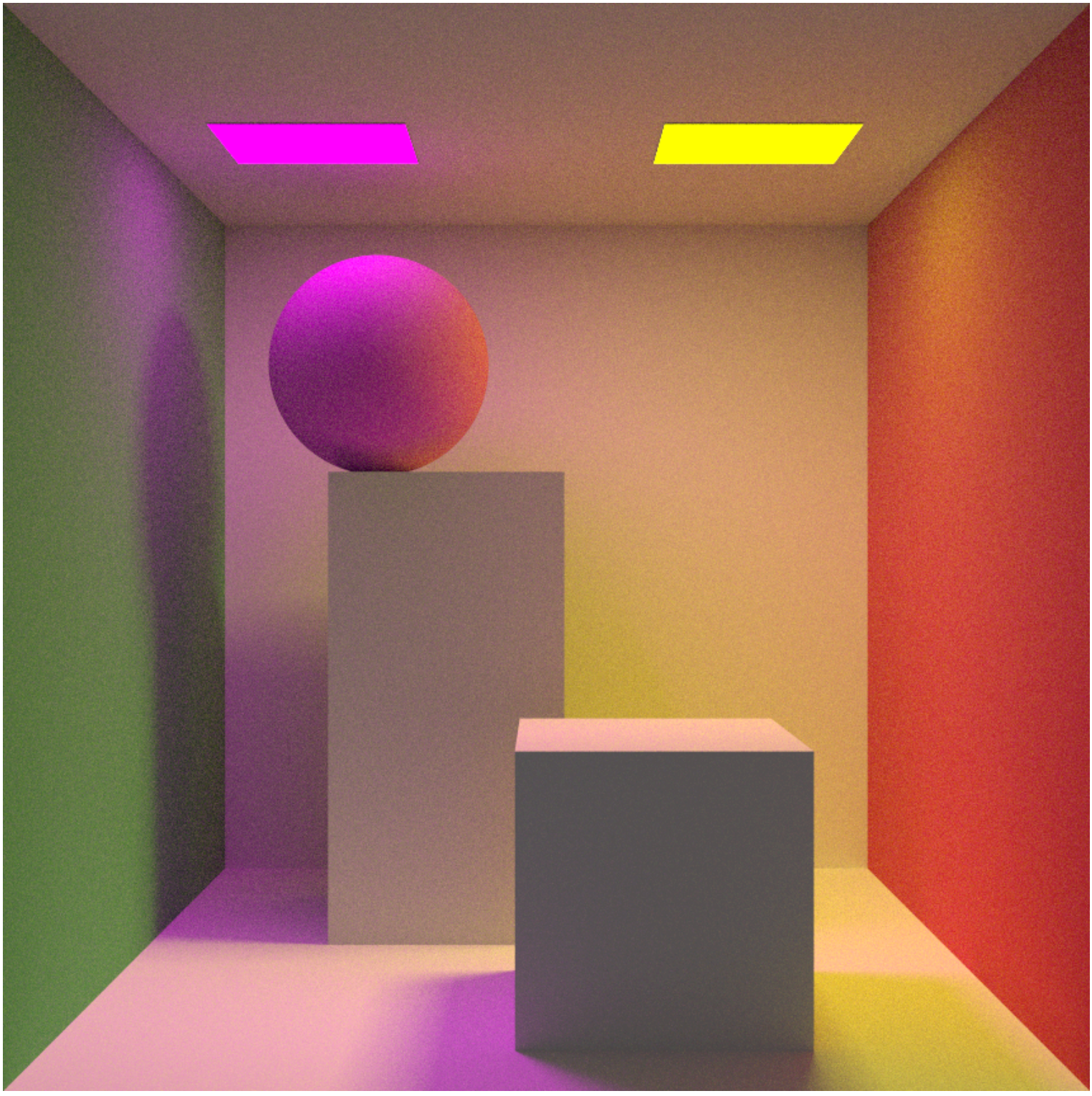
Path Tracer



Authors: Caio Valente, Eduardo Maksoud
Code: https://github.com/valentecaio/raytracing
Date: 2024-06-20

Introduction

This project was initiated during the "Fundamentals of Computer Graphics" master's class at PUC-Rio in 2024, under the guidance of Prof. Waldemar Celes. The development and execution of the project were carried out by the students Caio Valente and Eduardo Maksoud.

A Physics-Based Renderer is software designed to simulate the physical behaviour of light within a scene to generate realistic images. The renderer presented in this work employs the path tracing algorithm, which simulates the path of light rays as they interact with objects in the scene. This renderer determines the colour of each pixel in the image by tracing rays from the camera through the scene and calculating their interactions with objects. Path tracing enables the recursive casting of rays into the scene, allowing for the calculation of indirect lighting, thereby enhancing the realism of the rendered images.

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

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

The developed path 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 code

This project uses the header-only libraries GLM and TinyObjLoader, both 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-threaded mode can drastically reduce the time taken to render images. You can enable it by using the `make run_mt` command.

```
make         # compile code and generate binary in build/raytracer
make run     # run
make run_mt  # run in multi-thread using OpenMP
```

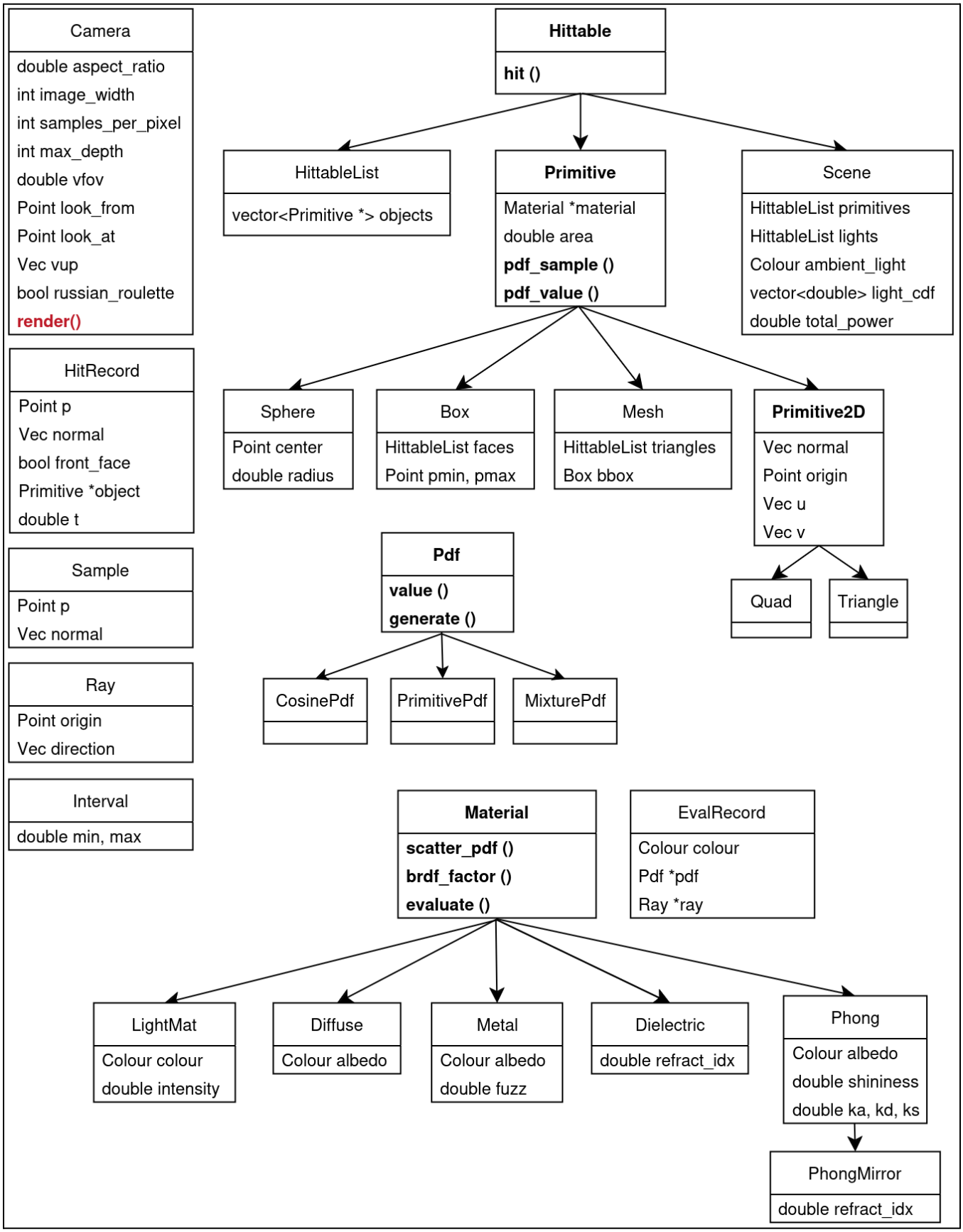
An `output.ppm` file will appear in the build folder.

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.

If you're using Windows, install MinGW and use Bash. The default Command Prompt commands may differ slightly from those shown above.

Code Structure

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



Class inheritance relationships in the path 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. To enable light sampling, the scene maintains a vector of doubles representing the light CDF function, which is updated whenever a new light is added. When sampling a new light object from the lights hittable-list, the index used is derived from the CDF sampling.

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, greatly affecting the performance of the algorithm and the quality of the final results.

Additionally, there is a russian roulette boolean parameter to enable Russian roulette for path termination. This method probabilistically terminates rays to avoid computing long paths with minimal contribution. If a path with a high termination probability continues, its contribution is divided by the probability to compensate for the higher likelihood of termination. Conversely, rays with a low termination probability are less contributing.

The camera also provides the entry point of the path 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.

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 pdf_sample() and pdf_value() 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.

Primitive classes are contained in the primitives/ directory.

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 primarily meant to work with the ray tracing algorithm and so do not return a ray to be traced next.

The Diffuse material is meant to work with a path tracing algorithm and in fact is the only material that has a scatter PDF function for the new ray to be traced and a BRDF ponderation. Materials such as Metal, and Dielectric are also meant to work with the path tracing algorithm but they do not use PDF sampling. All of them can be mixed with the Phong materials in the same scene without breaking the algorithm, but the results may not be accurate.

When evaluated, a diffuse material returns a CosinePdf to be used to trace the next ray in the path tracing algorithm. The Metal and Dielectric materials, on the other hand, return the only ray they can trace, in the direction of the reflection or refraction.

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, such as Box and Mesh, they are also treated as area light sources, but the results are inaccurate since sampling uniformly is not correctly implemented for them.

EvalRecord

Represents the information about the material evaluation at a hit point. It contains the evaluated colour of the material at the hit point and may contain a pointer to a pdf generator or a pointer to ray. When one of the two pointers exists, it means that a new ray should be cast, either directly to the ray parameter, or to a new ray obtained through the pdf parameter. If there is no pdf and no new ray, the ray was absorbed by the object. This approach makes this implementation a "hybrid" path tracer, enabling Phong materials to be mixed with Path Traced diffuse materials in the same scene.

LightMat and Phong materials absorb the ray and do not fill the pointers. They only return the evaluated colour.

Metal and Dielectric materials fill the ray pointer with a new ray in the direction of the evaluated reflection or refraction.

Diffuse materials fill the pdf pointer with a CosinePdf maxed at the normal of the hit point, which should be used to generate a new ray to be cast.

Sample

Represents a sample point on the surface of a primitive. It includes a point for the sample in global coordinates and a vector for the normal at that point. It is used to sample area lights for example.

CosinePdf

Represents a PDF for cosine-weighted sampling on a hemisphere. It inherits from the PDF class and is used to generate random directions weighted by the cosine of the angle with a given direction. This is the PDF definition that is used for diffuse materials.

Ray

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

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.

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 a pointer to the object that was hit.

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.

Utilities - utils, vec, random

The code also contains three utility namespaces: raytracer::utils, raytracer::vec and raytracer::random. The first contains utility functions for 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. The random namespace handles random number generation and sampling.
These utilities can be found in the utils/ directory.

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. When combined with a LightMat material, spheres can be used to create Point Lights.

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.

2D Primitives - Quad, Triangle

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 implements 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.

2D Primitives can be combined with the LightMat material to create area 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.

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.

Path Tracer Algorithm

```
Colour path_trace(Ray & ray) const {
  auto L = Colour(0); // accumulated radiance
  auto beta = Colour(1); // ponderation factor for the path

  for (int depth = 0; depth < max_depth; ++depth) {
    // try to hit an object in the scene, starting at 0.0001 to avoid self-intersection
    // misses are considered as ambient_light colour
    HitRecord hit;
    if (!scene.hit(ray, Interval(0.0001, infinity), hit)) {
      return L + beta * scene.ambient_light;
    }

    // HIT //
    // evaluate the material at the hit point
    auto mat = hit.object->material;
    EvalRecord eval = mat->evaluate(scene, ray, hit);

    // light source
    if (std::dynamic_pointer_cast < LightMat > (mat)) {
      // discard accidental light hits
      return (depth == 0) ? eval.colour : L;
    }

    // end path shooting a last ray to a light source
    Colour Le = scene.get_light_radiance(hit);
    L += Le * beta * eval.colour * mat->brdf_factor();

    if (eval.pdf) {
      // ray bounced and has a pdf (Diffuse)
      ray = Ray(hit.p, eval.pdf->generate());
      double pdf = eval.pdf->value(ray.direction());
      double scatter_pdf = mat->scatter_pdf(hit.normal(), ray);
      beta *= eval.colour * scatter_pdf / pdf;
    } else if (eval.ray) {
      // ray bounced and has a fixed direction (simple reflection)
      ray = * eval.ray;
      beta *= eval.colour;
    } else {
      // ray was absorbed (Phong materials)
      return eval.colour;
    }
  }
  return L;
}
```

camera.hpp

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

The path_trace() method contains the main logic of the path tracing algorithm.

As a parameter the method receives 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 accumulating the radiance (L) along the path of a ray and adjusting the path weight (beta) as the ray interacts with objects.

If the ray misses, it returns the ambient light colour, but if it hits an object the material gets evaluated. If the material is a light source, it only considers the light's colour if it's the first hit (depth 0); otherwise, it discards the hit. For diffuse materials, it generates a new ray direction using a PDF and adjusts the path weight accordingly. For reflective materials, it uses a fixed reflection direction. If the material absorbs the ray (like Phong materials), it returns the material's colour.

Get Light Radiance method

```
Colour get_light_radiance(const HitRecord& hit) const {
      // sample a light from the scene
      shared_ptr<Primitive> light = sample_light();
      auto lmat = std::static_pointer_cast<LightMat>(light->material);
      double pdf = lmat->intensity / total_power;

      // sample a point on the light source
      // spheres are point lights, other primitives are area lights
      Sample sample;
      Vec wi;
      auto point_light = std::dynamic_pointer_cast<Sphere>(light);

      if (point_light) {
        sample.p = point_light->center;
        wi = glm::normalize(sample.p - hit.p);
        sample.normal = -wi;
      } else {
        sample = light->pdf_sample();
        wi = glm::normalize(sample.p - hit.p);
      }

      auto ray = Ray(hit.p, wi);

      // launch ray and try to hit the light source
      HitRecord hitrec;
      if (this->hit(ray, Interval(0.0001, infinity), hitrec) && hitrec.object == light) {
        double distance = glm::length(sample.p - hitrec.p);
        double cos1 = std::max(glm::dot(hit.normal(), wi), 0.0);
        double cos2 = std::max(glm::dot(-wi, saGmple.normal), 0.0);
        Colour radiance = lmat->radiance(distance);
        return radiance * cos1 * cos2 / pdf;
      } else {
        return Colour(0);
      }
    }
```

The get_light_radience method is used to sample a light source from the scene and calculate its impact on a point hit by a ray. First it selects a light source using a CDF based on the power of the lights. For point lights (like spheres), it uses the centre of the sphere, while for other types of lights, it samples a point on their surface. A ray is then cast from the hit point to this sampled light point. If the ray directly hits the light source, it calculates the light contribution using the angles between the ray and the surface normals, along with the light's distance, intensity and colour.

Sample Hemisphere Cosine method

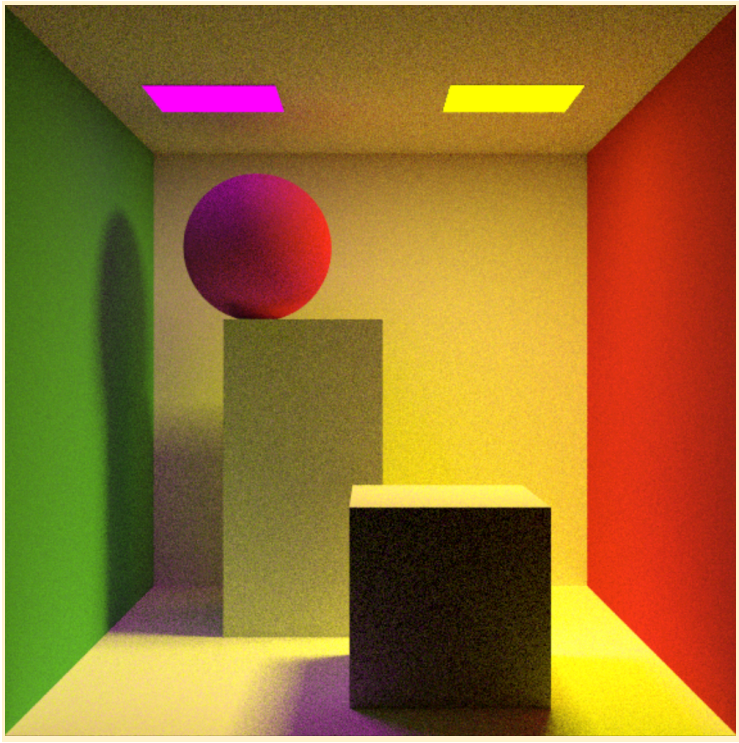
```
inline Vec sample_hemisphere_cosine() {
  double r   = random::rand();          // r   = random in [0, 1)
  double phi = random::rand(0, 2*M_PI); // phi = random in [0, 2pi)
  auto x = cos(phi)*sqrt(r);            // x = sqrt(r) * cos(phi)
  auto y = sin(phi)*sqrt(r);            // y = sqrt(r) * sin(phi)
  auto z = sqrt(1-r);                   // z = sqrt(1 - r)
  return Vec(x, y, z);
}

inline Vec sample_hemisphere_cosine(const Vec& normal) {
  Vec vec = sample_hemisphere_cosine();
  return glm::normalize(vec::change_basis(normal, vec))
}
```

These two methods generate new ray directions for diffuse materials in the path tracing algorithm. The first method, sample_hemisphere_cosine(), creates a random direction in a hemisphere using a cosine-weighted distribution. The second method, sample_hemisphere_cosine(const Vec& normal), takes this direction and adjusts it to align with the surface normal at the hit point, transforming it from local to global space.

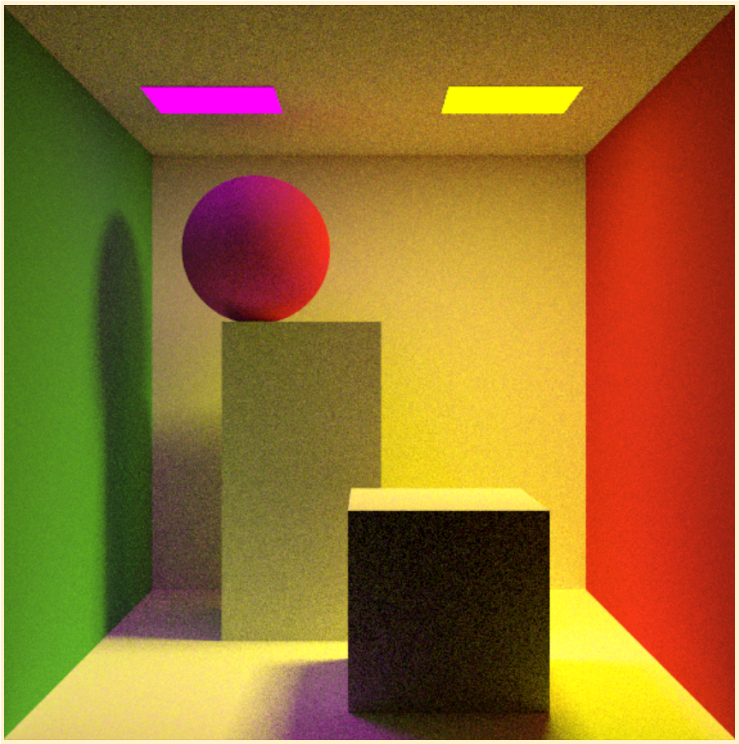
The Russian Roulette technique

```
if (russian_roulette && depth > min_depth) {
  // continuation probability (at least 10%)
  double p = utils::max({beta.r, beta.g, beta.b, 0.1});
  if (random::rand() > p) {
    return L;
  }
  beta /= p;
}
```

The Russian Roulette method is used to probabilistically terminate paths to improve efficiency. After a certain depth, the continuation of a ray is decided based on a probability derived from the ray's importance factor (beta). If the random value exceeds this probability, the path is terminated; otherwise, the path continues, and the importance factor is adjusted. This technique helps reduce computation time while maintaining the quality of the rendered image. Comparing images rendered with and without Russian Roulette will demonstrate that while the final results are similar, the method with Russian Roulette is more computationally efficient. The scenes below were rendered with 49 samples per pixel using a max depth of 20.



(A) Scene rendered without Russian Roulette (586 seconds)

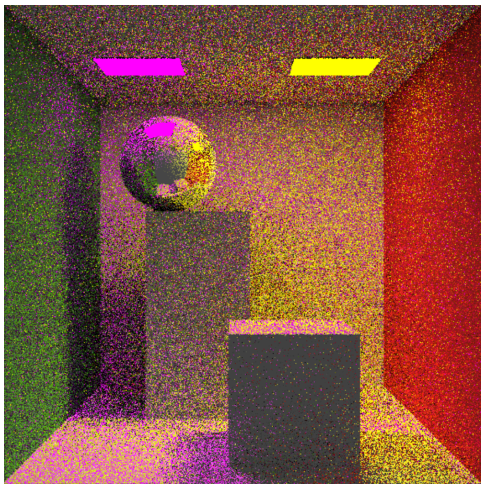


(B) Scene rendered with Russian Roulette (344 seconds)

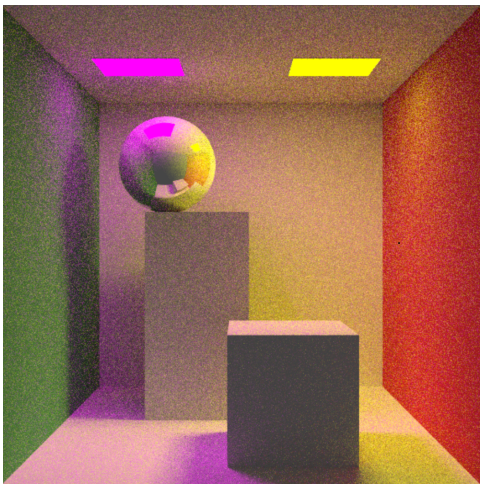
The Pixel Sampling Trade-off

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.

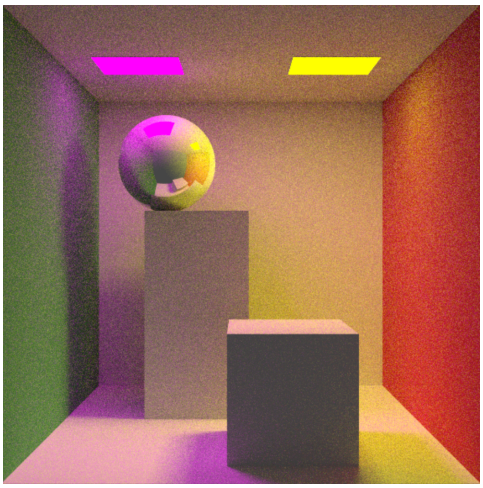
A pixel is a point, but in the context of computer graphics, it is a square area in the image plane. By uniformly sampling rays in the pixel area, we can estimate the final colour of the whole area by averaging the colours of the rays that hit the scene.



(A) 1 sample per pixel (7 seconds)



(B) 25 samples per pixel (165 seconds)



(C) 49 samples per pixel (323 seconds)

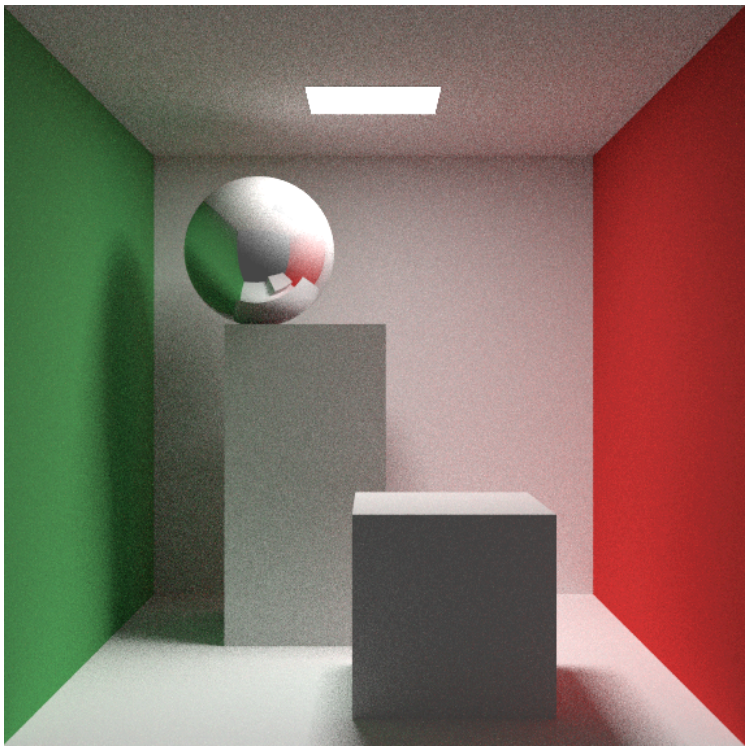
Lights: Point Lights and Area Lights

Lights consist of a Primitive with the LightMat material. However, to accurately represent light, a primitive needs to be able to be randomly sampled following a PDF. This can be achieved by implementing the pdf_sample() and pdf_value() methods, which is not yet done for all Primitives.

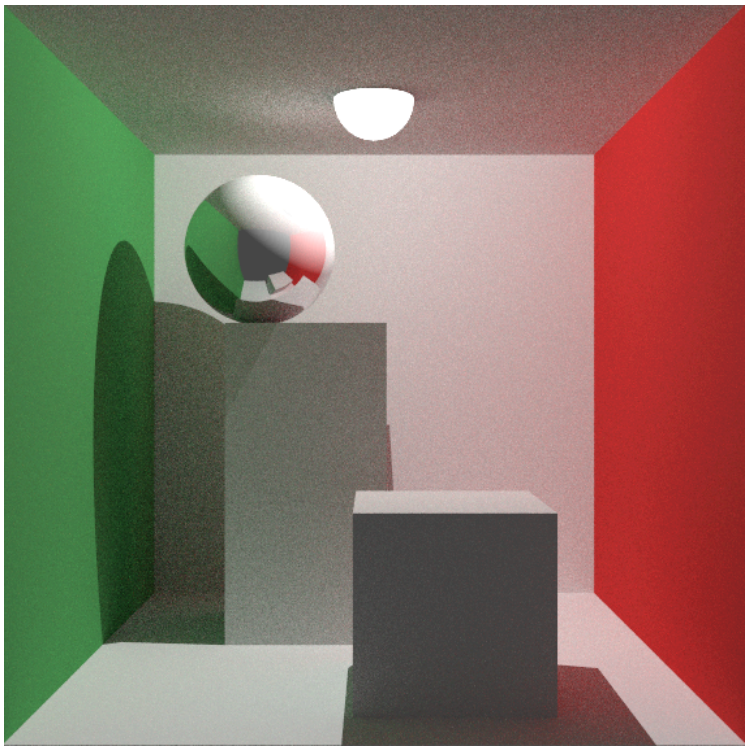
Quads and triangles implement the necessary methods and can be used as area lights.

Boxes and meshes can also be used as area lights, but are inaccurate.

Spheres are handled as a special case and are considered point lights.



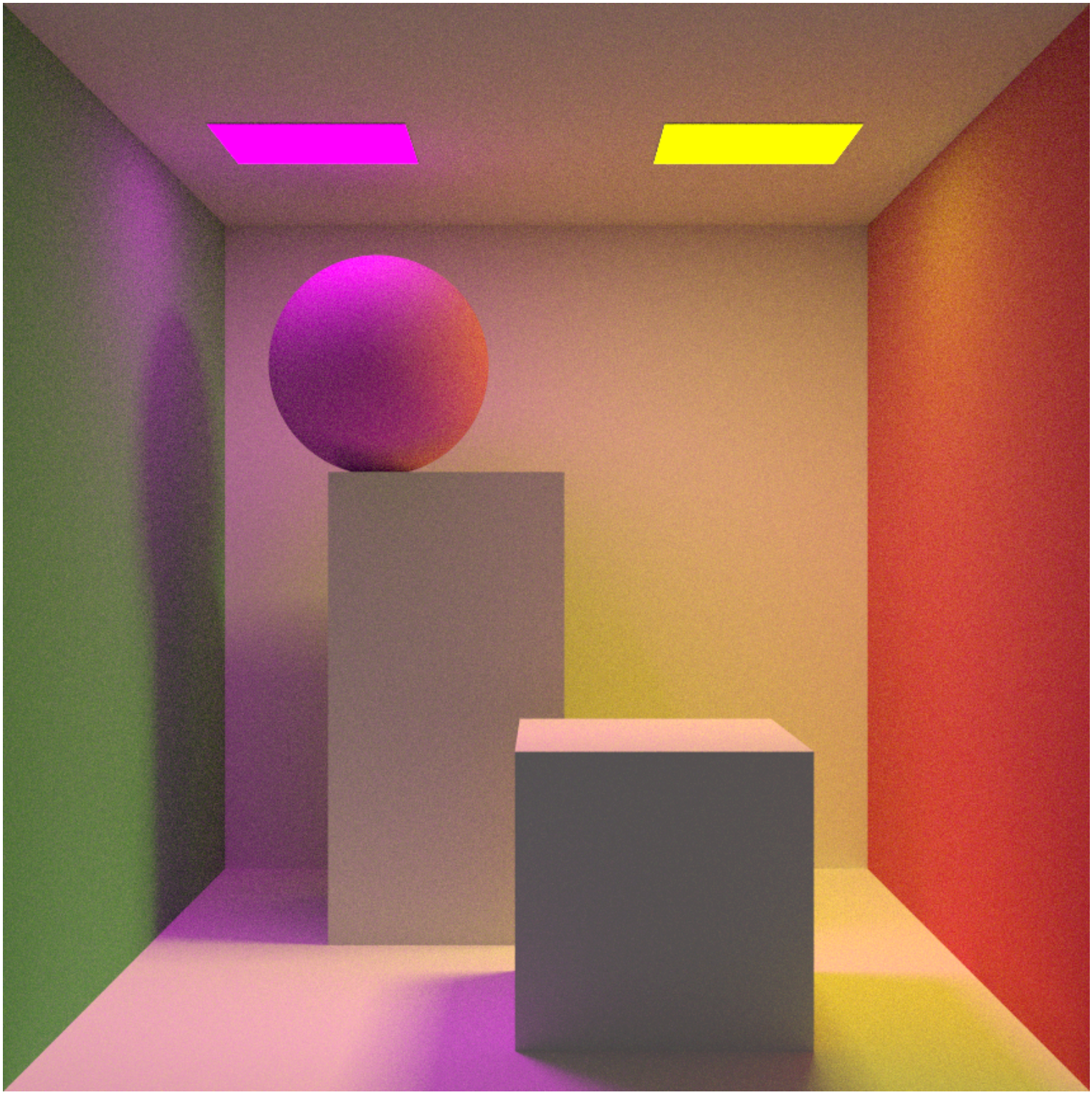
(A) Area light scene



(B) Point light scene

Results

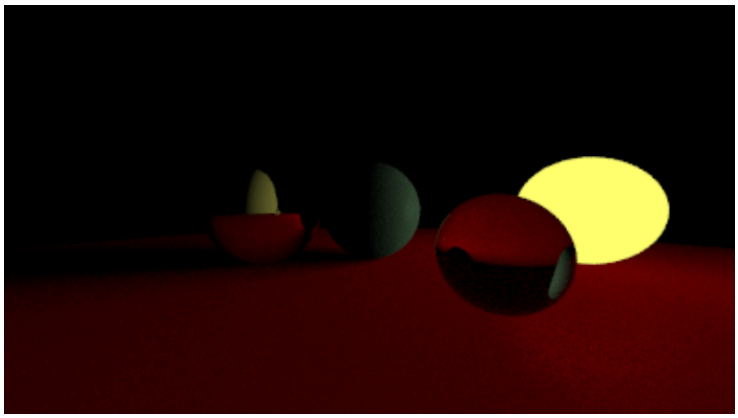
Cornell Box - Diffuse



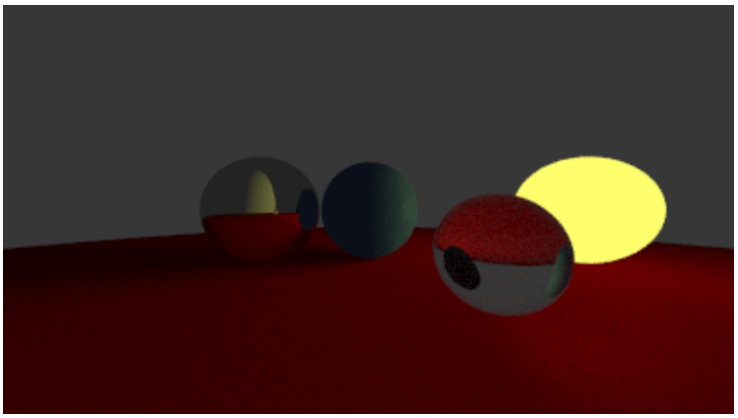
Samples per pixel: 200.
Resolution: 900 x 900 pixels.
Path lengths: 4 to 20 rays with russian roulette.
Time: 10 minutes (12 threads, i5-12450H CPU).

```
        if (!scene.hit(ray, Interval(0.0001, infinity), hit)) {
          // miss
          return L + beta * scene.ambient_light;
        }
```

If a ray doesn't hit any objects, the scene's background colour is determined by the accumulated radiance (L), combined with ambient light multiplied by the beta factor. If we set the ambient light to (0,0,0), there's no extra light added for rays that miss objects. Therefore, the background colour depends entirely on the accumulated radiance L from other light interactions within the scene. If L does not capture contributions from indirect light sources, the background will appear black or very dark, depending on the intensity of L. In the first image below that is exactly the case, resulting in a much darker scene.



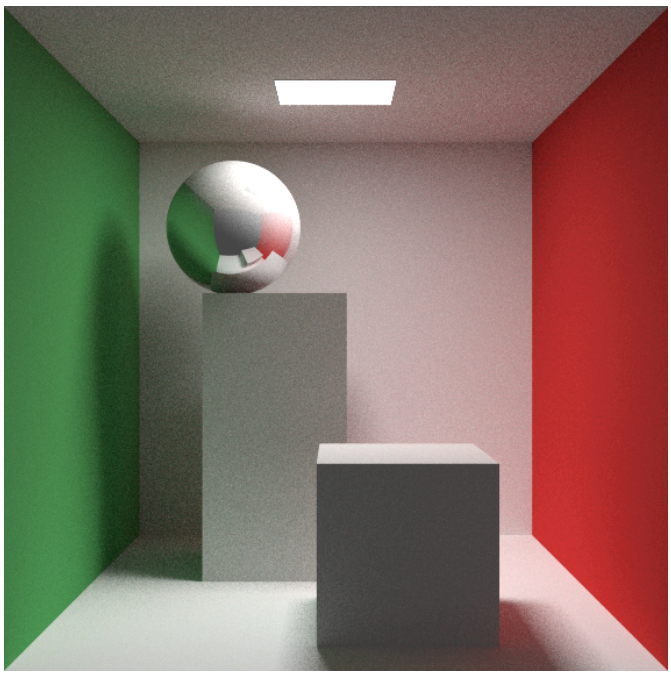
(A) Scene without ambient light



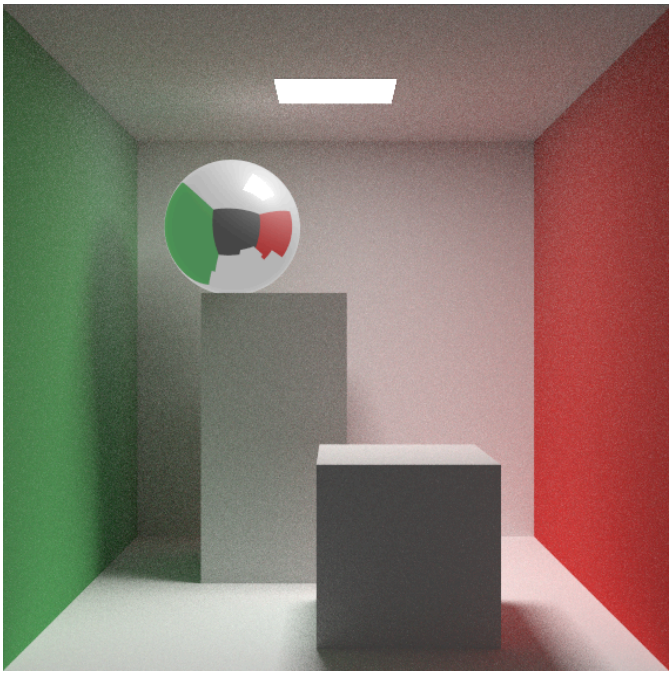
(B) Scene with ambient light

Hybrid Path Tracing

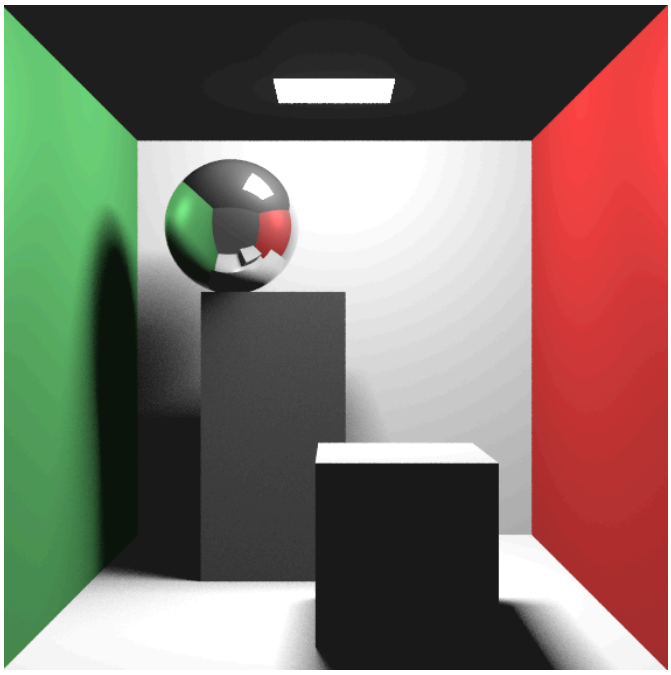
The four images below show different rendering techniques with varying results. The first image, fully path traced, is the best, showing bright, well-lit scenes where light bounces off surfaces, including the ceiling. This technique also clearly defines the corners of the walls, making them much more visible. The second image uses a Phong mirror sphere and looks odd because the reflections don’t match the path traced diffuse materials from the walls. The third image is purely ray traced with Phong materials and looks much darker and rougher, with a completely dark ceiling and deeper shadows, and the corners of the walls are less defined. The fourth image, with Phong walls and a metal sphere, looks better than the second, offering improved reflections though it still doesn't match the quality and clarity seen in the first image.



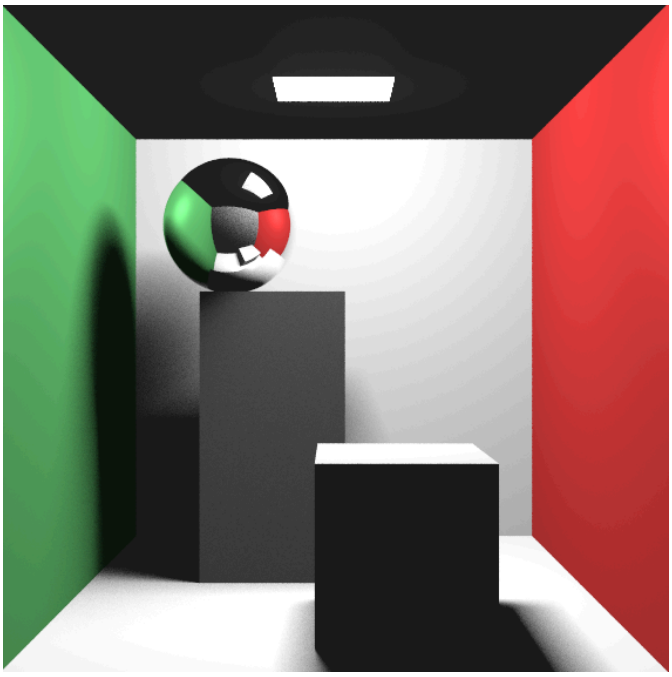
(A) A pure path tracing scene (Diffuse)



(B) Hybrid scene (Diffuse + PhongMirror)



(C) A pure ray tracing scene (Phong)



(D) Hybrid scene (Phong + Metal)

Assignment of Tasks

For this assignment, we built upon Caio's previous work and introduced new features. Together, we developed the **path_trace()** and **get_light_radience()** methods, strongly based on the pseudo-code proposed by prof. Waldemar, with Caio focusing on the PDF sampling (CosinePdf class) and the diffuse material with BRDF, while Eduardo implemented the hemisphere sampling method and the russian roulette feature. Additionally, both collaborated on the production of the final report.

References

Books *"Ray Tracing in One Weekend Book Series"*
By Peter Shirley, Trevor David Black, Steve Hollasch
Available at https://raytracing.github.io

Book *"Physically Based Rendering: From Theory To Implementation"*
By Matt Pharr, Wenzel Jakob, Greg Humphreys
Available at https://www.pbr-book.org