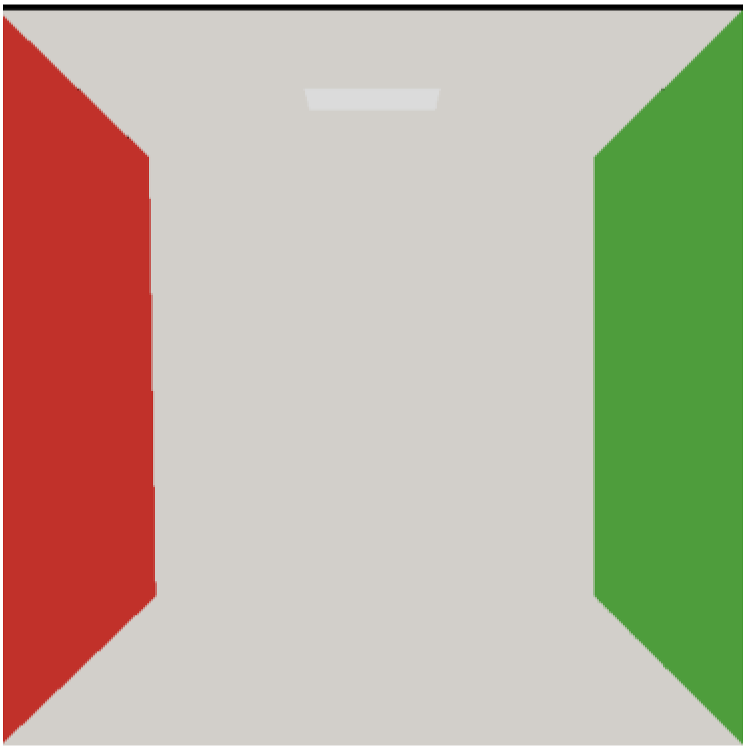
EECS598 HW2

author: Yilin Jia

Usage

```
g++ -std=c++17 -O3 -o rayTracing *.cpp
./rayTracing
```

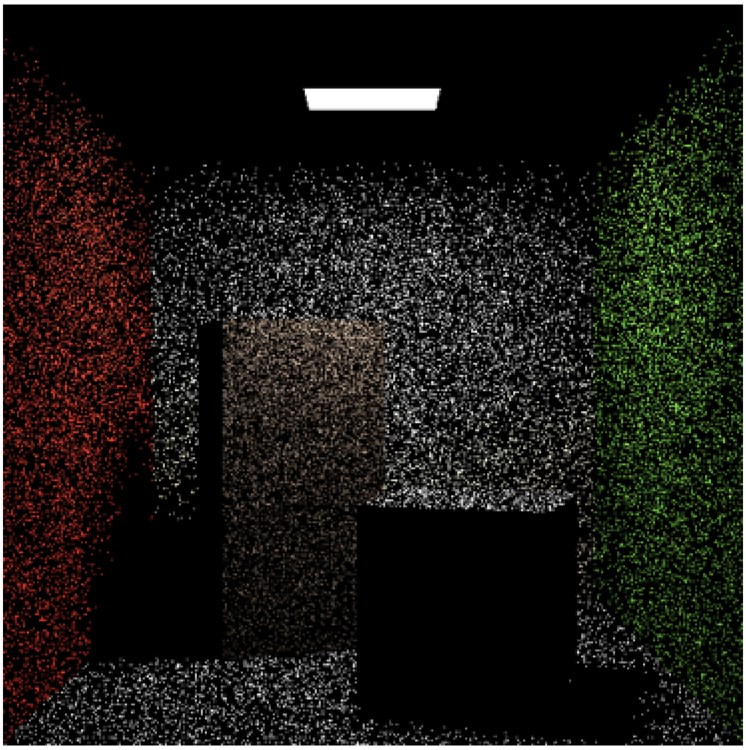
task 5.3 Basic



```
Vec3 Scene::trace(const Ray &ray, int bouncesLeft, bool discardEmission) {
    if constexpr(DEBUG) {
        assert(ray.isNormalized());
    }

    // Use the getIntersection function to find the closest intersection
    Intersection intersection = getIntersection(ray);

    // If there was an intersection, return the diffuse color of the surface
    if (intersection.happened) {  // Assuming intersection.hit tells whether
there's a valid hit
        return intersection.getDiffuseColor();  // Return the diffuse color of
the hit object
    }

    // If no intersection, return a background color (e.g., black)
    return Vec3(0.0f, 0.0f, 0.0f);  // Return black color as the default
background
```

```
}
```

task 6.4 Direct Illumination

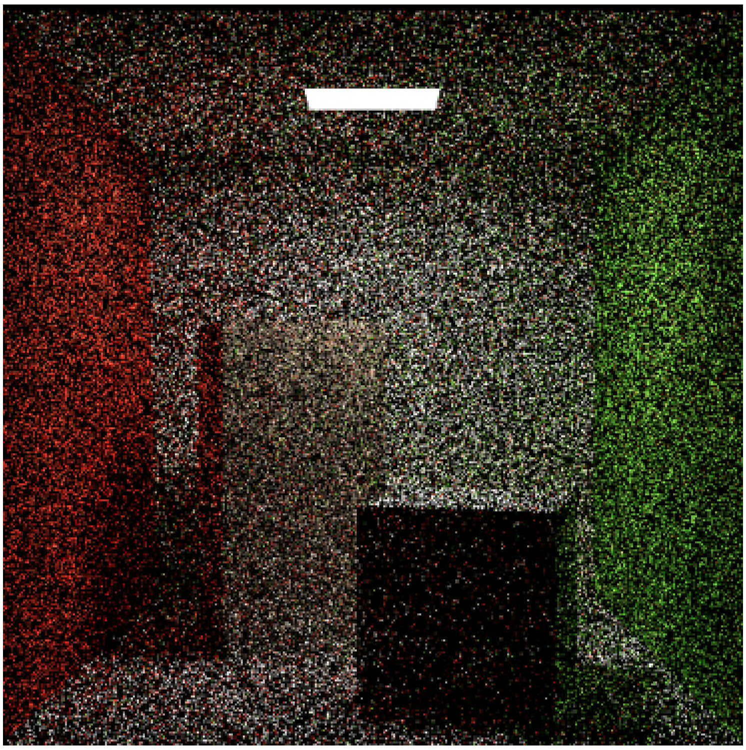


```
Vec3 Scene::trace(const Ray &ray, int bouncesLeft, bool discardEmission) {
    if constexpr (DEBUG) {
        assert(ray.isNormalized());
    }

    // Use the getIntersection function to find the closest intersection
    Intersection intersection = getIntersection(ray);
    if (!intersection.happened) {
        // If no intersection, return background color (black in this case)
        return Vec3(0.0f, 0.0f, 0.0f);
    }

    // Direct lighting computation starts here
    Vec3 Lo(0.0f, 0.0f, 0.0f);

    // Get emitted radiance from the intersected object (if emissive)
    Vec3 Le = intersection.getEmission();

    if (!discardEmission && Le.getLength() > 0.0f) {
        // Add emission directly if it's not discarded
        Lo += Le;
    }

    // Sample a random direction on the hemisphere oriented around the normal
    Vec3 normal = intersection.getNormal();
    Vec3 wi = Random::randomHemisphereDirection(normal);

    // Trace a ray in the sampled direction
```

```
    Ray shadowRay(intersection.pos, wi);
    Intersection secondIntersection = getIntersection(shadowRay);

    if (secondIntersection.happened) {
        // Incoming radiance from the light source or another object
        Vec3 Li = secondIntersection.getEmission(); // Assuming the emission of
intersected object is Li

        // Calculate BRDF at the intersection point
        Vec3 brdf = intersection.calcBRDF(-shadowRay.dir, -ray.dir); // wi and wo
are reversed because we trace backward

        // Calculate the cosine term (n · wi)
        float cosineTerm = std::max(0.0f, normal.dot(wi));

        // Assume uniform sampling over the hemisphere for pdf
        float pdf = 1.0f / (2.0f * PI);

        // Apply the direct illumination equation
        Lo += (Li * brdf * cosineTerm) / pdf;
    }

    // Return the outgoing radiance
    return Lo;
}
```

task 7.1 Global Illumination

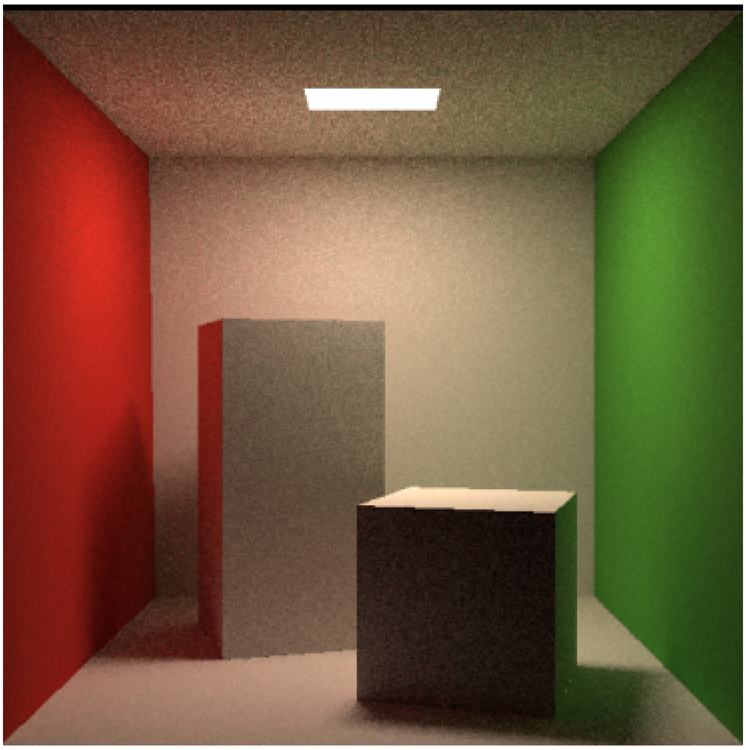
Render the image with exactly 32 SPP, and MAX DEPTH at least 4



```
if constexpr (DEBUG) {
        assert(ray.isNormalized());
    }
```

```
    // Base case: stop recursion when no bounces are left
    if (bouncesLeft <= 0) {
        return Vec3(0.0f, 0.0f, 0.0f);  // Return zero radiance for no more
bounces
    }

    // Use the getIntersection function to find the closest intersection
    Intersection intersection = getIntersection(ray);
    if (!intersection.happened) {
        // If no intersection, return background color (black in this case)
        return Vec3(0.0f, 0.0f, 0.0f);
    }

    // Direct lighting computation starts here
    Vec3 Lo(0.0f, 0.0f, 0.0f);

    // Get emitted radiance from the intersected object (if emissive)
    Vec3 Le = intersection.getEmission();

    if (!discardEmission) {
        // Add emission directly if it's not discarded
        Lo += Le;
    }

    // Sample a random direction on the hemisphere oriented around the normal
    Vec3 normal = intersection.getNormal();
    Vec3 wi = Random::randomHemisphereDirection(normal);

    // Trace a ray in the sampled direction (this will handle indirect lighting)
    Ray nextRay(intersection.pos, wi);

    // Recursively trace the next ray for indirect lighting contribution
    Vec3 indirectRadiance = trace(nextRay, bouncesLeft - 1, discardEmission);

    // Calculate BRDF at the intersection point
    Vec3 brdf = intersection.calcBRDF(-nextRay.dir, -ray.dir); // wi and wo are
reversed because we trace backward

    // Calculate the cosine term (n · wi)
    float cosineTerm = std::max(0.0f, normal.dot(wi));

    // Assume uniform sampling over the hemisphere for pdf
    float pdf = 1.0f / (2.0f * PI);

    // Add indirect illumination to the outgoing radiance
    Lo += (indirectRadiance * brdf * cosineTerm) / pdf;

    // Return the outgoing radiance
    return Lo;
}
```

task 8.3 Acc



```
Vec3 Scene::trace(const Ray &ray, int bouncesLeft, bool discardEmission) {
    if constexpr (DEBUG) {
        assert(ray.isNormalized());
    }

    // Base case: stop recursion when no bounces are left
    if (bouncesLeft <= 0) {
        return Vec3(0.0f, 0.0f, 0.0f);  // Return zero radiance if no more
bounces
    }

    // Find intersection of the ray with the scene
    Intersection intersection = getIntersection(ray);
    if (!intersection.happened) {
        // If no intersection, return background color (black in this case)
        return Vec3(0.0f, 0.0f, 0.0f);
    }

    // Initialize the outgoing radiance
    Vec3 Lo(0.0f, 0.0f, 0.0f);

    // Get emitted radiance from the intersected object (if emissive)
    Vec3 Le = intersection.getEmission();
    if (!discardEmission) {
        // Add emission directly if not discarded
        Lo += Le;
    }

    // Handle direct lighting by sampling the light source
    Intersection lightSample = sampleLight();
    Vec3 lightDir = lightSample.pos - intersection.pos;
```

```
    float distanceToLight = lightDir.getLength();
    lightDir.normalize();

    // Create a shadow ray towards the light source
    Ray rayToLight(intersection.pos, lightDir);

    // Check if the light is visible (not blocked by any objects)
    Intersection shadowIntersection = getIntersection(rayToLight);
    if (shadowIntersection.happened &&
shadowIntersection.getEmission().getLength()!=0) {
        // If the light is not blocked, calculate direct radiance contribution
        Vec3 lightRadiance = lightSample.getEmission();
        Vec3 brdf = intersection.calcBRDF(-lightDir, -ray.dir);  // wi and wo are
reversed
        float cosTheta = std::max(0.0f, intersection.getNormal().dot(lightDir));
        float cosThetaLight = std::max(0.0f, -
lightSample.getNormal().dot(lightDir));
        float pdfLightSample = 1.0f / lightArea;  // Access the precomputed light
area

        // Add direct radiance contribution to the total outgoing radiance
        Lo += (lightRadiance * brdf * cosTheta * cosThetaLight) /
              (distanceToLight * distanceToLight * pdfLightSample);
    }

    // Handle indirect lighting using importance sampling (cosine-weighted
hemisphere)
    Vec3 normal = intersection.getNormal();
    Vec3 wi = Random::cosWeightedHemisphere(normal);  // Sample a direction

    // Probability density function for cosine-weighted sampling
    float pdf = normal.dot(wi) / PI;

    // Trace a ray in the sampled direction (indirect radiance)
    Ray nextRay(intersection.pos, wi);
    Vec3 indirectRadiance = trace(nextRay, bouncesLeft - 1, true);  //
Recursively trace

    // Calculate the BRDF and cosine term
    Vec3 brdf = intersection.calcBRDF(-wi, -ray.dir);  // wi and wo are reversed
    float cosineTerm = std::max(0.0f, normal.dot(wi));

    // Add indirect radiance contribution to the total outgoing radiance
    Lo += (indirectRadiance * brdf * cosineTerm) / pdf;

    // Return the total outgoing radiance (direct + indirect)
    return Lo;
}
```

Feature

Multi-threading acceleration.

Physically-based Rendering (PBR).