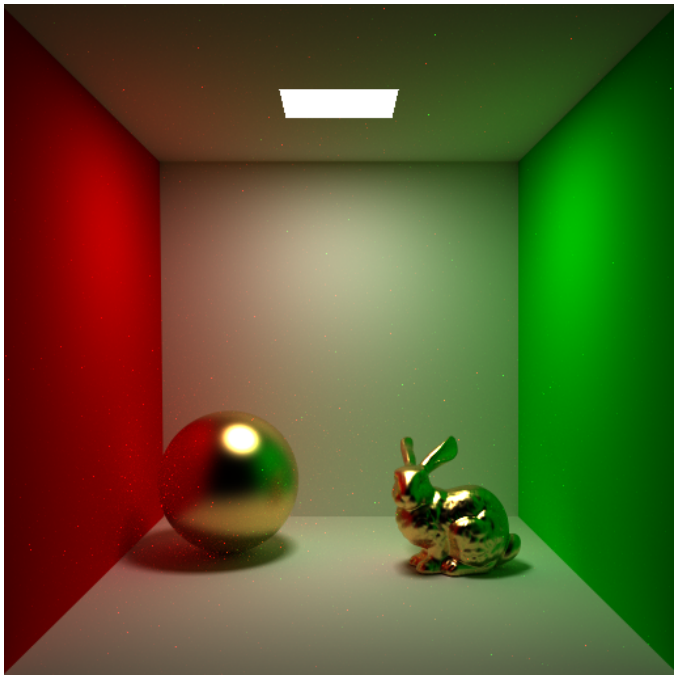
PHYSICALLY BASED RENDERING

KIM HARDER FOG - S070808
PETER BAY BASTIAN - S113119



Lab journal for 02941 Physically Based Rendering
Technical University of Denmark

June 2016

CONTENTS

INTRODUCTION

For the first 4 weeks we've added renderlogs to the images, but after implementing pathtracing we sort of forgot to log the render times but we've added information from the `'b'` output where we remembered to save it.

The work has been equally distributed.

1

WORKSHEET 1 DELIVERABLES

We've loaded the Stanford bunny into the path tracer. The path must be passed in as command line arguments relatively to where the compiled executable is run from, e.g. `../models/bunny.obj` to load the bunny model. The pre-visualization along with render log can be seen in figure 1.

We've implemented the Lambertian shading model along with a directional light. The resulting render and render log can be seen in figure 2.

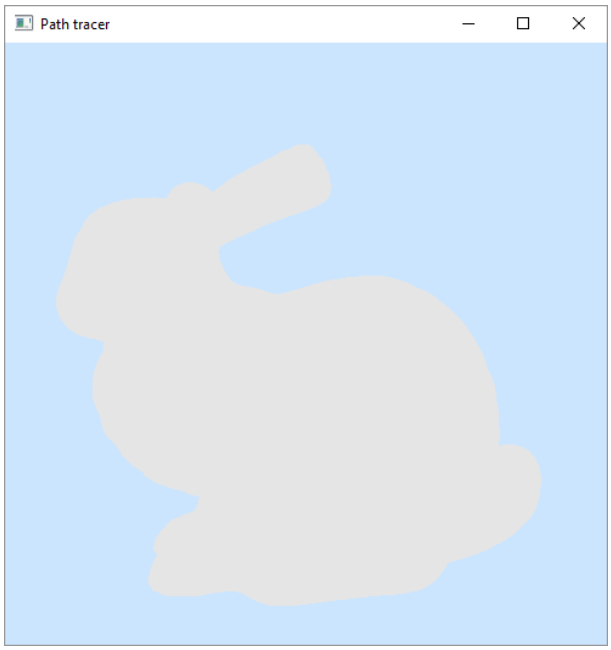
Then we adjusted the `.mtl`-file by adding a seperate `illum 0` material called `pink`, which we modified `bunny.obj` to use instead of `white`. Parameters can be seen in listing 1 and a render of the bunny with the new material and the associated render log can be seen in figure 3.

Listing 1: Additions to bunny.mtl

```
newmtl pink
Ka 0.0 0.0 0.0
Kd 1.0 0.75 0.80
Ks 1.0 1.0 1.0
illum 0
```

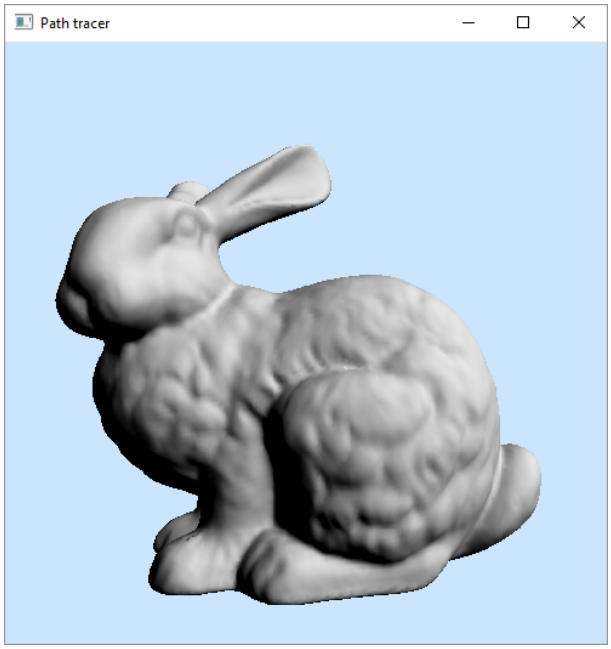
Afterwards we implemented hard shadows and used extra jitter samples to get nicely anti-aliased lines. This can be seen in figure 4. Jitter sampling works by sampling multiple times at randomized position inside each pixel, and then taking the averaging of all the samples.

Finally we loaded the Cornell box scene (consisting of `CornellBox.obj` and `CornellBlocks.obj`). This scene contains an area light, which we've implemented in a simplified manner, by only sampling the center of the area. See figure 5. Afterwards we implemented the hard shadows in the same way as with the directional light, figure 6.



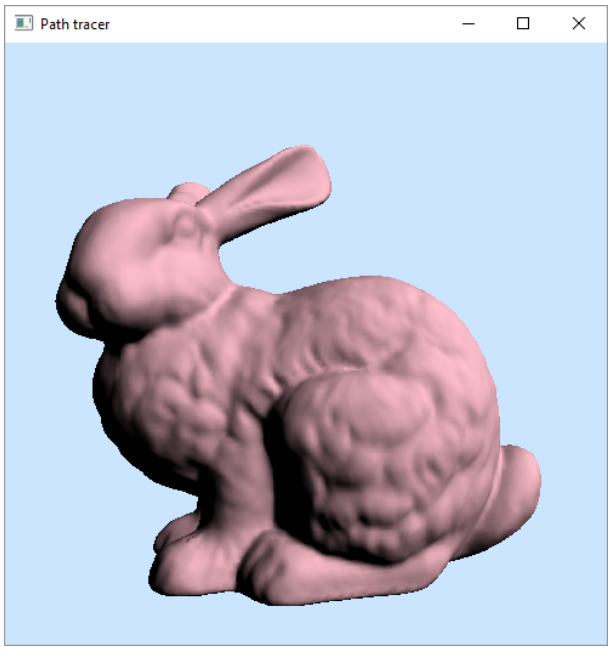

```
Loading ../models/bunny.obj
Computing normals
No. of triangles: 69451
Loading ..\models\media.mpml
Background ambient: [ 0.8 0.9 1 ]
Adding default light: Directional light (emitted
    radiance [ 3.14159 3.14159 3.14159 ], direction [
     -0.57735 -0.57735 -0.57735 ]).
Building acceleration structure...(time: 0.503)
Building photon maps...
Particles in caustics map: 0 (photons shot: 1)
Particles in global map: 0 (photons shot: 1)
Building time: 0.004
Generating scene display list..........
```

Figure 1: Pre visualization of Stanford bunny



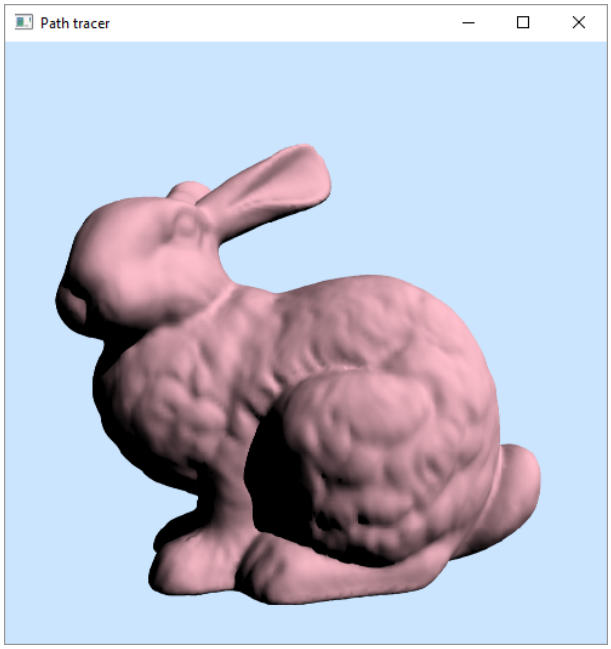

```
Loading ../models/bunny.obj
Computing normals
No. of triangles: 69451
Loading ..\models\media.mpml
Background ambient: [ 0.8 0.9 1 ]
Adding default light: Directional light (emitted
    radiance [ 3.14159 3.14159 3.14159 ], direction [
     -0.57735 -0.57735 -0.57735 ]).
Building acceleration structure...(time: 0.582)
Building photon maps...
Particles in caustics map: 0 (photons shot: 1)
Particles in global map: 0 (photons shot: 1)
Building time: 0.004
Generating scene display list..........
Switched to shader number 1
Generating scene display list..........
Raytracing.......... - 0.168 secs
```

Figure 2: Stanford bunny illuminated by a directional light using a Lambertian shading model




```
Loading ../models/bunny.obj
Computing normals
No. of triangles: 69451
Loading ..\models\media.mpml
Background ambient: [ 0.8 0.9 1 ]
Adding default light: Directional light (emitted
    radiance [ 3.14159 3.14159 3.14159 ], direction [
     -0.57735 -0.57735 -0.57735 ]).
Building acceleration structure...(time: 0.557)
Building photon maps...
Particles in caustics map: 0 (photons shot: 1)
Particles in global map: 0 (photons shot: 1)
Building time: 0.005
Generating scene display list..........
Switched to shader number 1
Generating scene display list..........
Raytracing.......... - 0.17 secs
```

Figure 3: Stanford bunny illuminated by a directional light using a Lambertian shading model, with a custom pink material




```
Loading ../models/bunny.obj
Computing normals
No. of triangles: 69451
Loading ..\models\media.mpml
Background ambient: [ 0.8 0.9 1 ]
Adding default light: Directional light (emitted
    radiance [ 3.14159 3.14159 3.14159 ], direction [
     -0.57735 -0.57735 -0.57735 ]).
Building acceleration structure...(time: 0.526)
Building photon maps...
Particles in caustics map: 0 (photons shot: 1)
Particles in global map: 0 (photons shot: 1)
Building time: 0.005
Generating scene display list..........
Rays per pixel: 4
Rays per pixel: 9
Rays per pixel: 16
Switched to shader number 1
Generating scene display list..........
Raytracing.......... - 4.043 secs
```

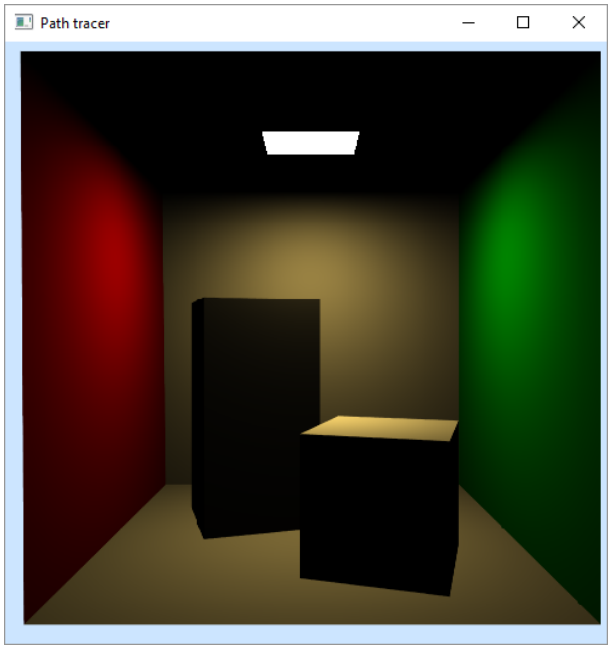
Figure 4: Stanford bunny illuminated by a directional light using a Lambertian shading model with hard shadows and 16 samples per pixel antialiasing

Listing 2: Directional.cpp

```
bool Directional::sample(const Vec3f& pos, Vec3f& dir,
    Vec3f& L) const
{
  dir = -light_dir;
  L = emission;

  if (shadows) {
    Ray shadow_ray(pos, dir);
    tracer->trace(shadow_ray);
    return !shadow_ray.has_hit;
  }

  return true;
}
```

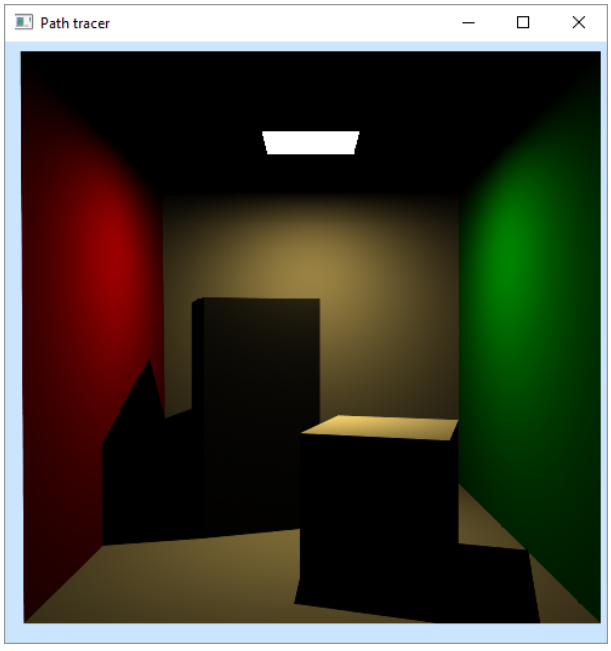
Listing 3: Lambertian.cpp

```
Vec3f Lambertian::shade(Ray& r, bool emit) const
{
  Vec3f rho_d = get_diffuse(r);
  Vec3f result(0.0f);

    Vec3f dir, L;
  for (auto light : lights) {
    int no_of_samples = light->get_no_of_samples();
    for (int i = 0; i < no_of_samples; i++) {
      if (light->sample(r.hit_pos, dir, L)) {
        result += L * fmax(0.0f, dot(dir, r.hit_normal));
      }
    }
    result != no_of_samples;
  }

  result *= rho_d * M_1_PIf;

  return result + Emission::shade(r, emit);
}
```




```
Loading ../models/CornellBox.obj
Computing normals
No. of triangles: 12
Loading ../models/CornellBlocks.obj
Computing normals
No. of triangles: 24
Loading ..\models\media.mpml
Background ambient: [ 0.8 0.9 1 ]
Building acceleration structure...(time: 0.008)
Building photon maps...
Particles in caustics map: 0 (photons shot: 1)
Particles in global map: 0 (photons shot: 1)
Building time: 0.005
Generating scene display list
Toggled shadows off
Generating scene display list
Switched to shader number 1
Generating scene display list
Rays per pixel: 4
Rays per pixel: 9
Rays per pixel: 16
Raytracing.......... - 5.096 secs
```

Figure 5: Cornell box with area light




```
Loading ../models/CornellBox.obj
Computing normals
No. of triangles: 12
Loading ../models/CornellBlocks.obj
Computing normals
No. of triangles: 24
Loading ..\models\media.mpml
Background ambient: [ 0.8 0.9 1 ]
Building acceleration structure...(time: 0.005)
Building photon maps...
Particles in caustics map: 0 (photons shot: 1)
Particles in global map: 0 (photons shot: 1)
Building time: 0.008
Generating scene display list
Rays per pixel: 4
Rays per pixel: 9
Rays per pixel: 16
Switched to shader number 1
Generating scene display list
Raytracing.......... - 11.009 secs
```

Figure 6: Cornell box with area light and shadows

Listing 4: AreaLight.cpp

```
bool AreaLight::sample(const Vec3f& pos, Vec3f& dir, Vec3f&
    L) const
{
  // Get geometry info
  const IndexedFaceSet& geometry = mesh->geometry;
  const IndexedFaceSet& normals = mesh->normals;
  const float no_of_faces = geometry.no_faces();
  const float no_of_vertices = geometry.no_vertices();

  Vec3f center_normal(0.0f);
  Vec3f center_pos(0.0f);
  Vec3f Le_A(0.0f);

  for (size_t i = 0; i < no_of_faces; ++i) {
    auto face = geometry.face(i);

    // Vertices
    auto x0 = geometry.vertex(face[0]);
    auto x1 = geometry.vertex(face[1]);
    auto x2 = geometry.vertex(face[2]);

    // Edges
    auto e1 = x1 - x0;
    auto e2 = x2 - x0;

    auto L_e = get_emission(i);
    auto A_i = 0.5f * cross(e1, e2).length();

    Le_A += A_i * L_e;

    center_pos += x0 + x1 + x2;
    center_normal += normals.vertex(face[0]) + normals.
        vertex(face[1]) + normals.vertex(face[2]);
  }

  center_normal /= no_of_vertices;
  center_pos /= no_of_vertices;

  auto d = center_pos - pos;
  dir = normalize(d);

  float cos_theta = dot(-dir, center_normal);
  float r_sqr = d.length() * d.length();

  L = Le_A * cos_theta / r_sqr;

  if (shadows) {
    Ray shadow_ray(pos, dir, 1.0e-4f, d.length() - 1.0e-4f)
        ;
    tracer->trace(shadow_ray);
    return !shadow_ray.has_hit;
  }

  return true;
}
```

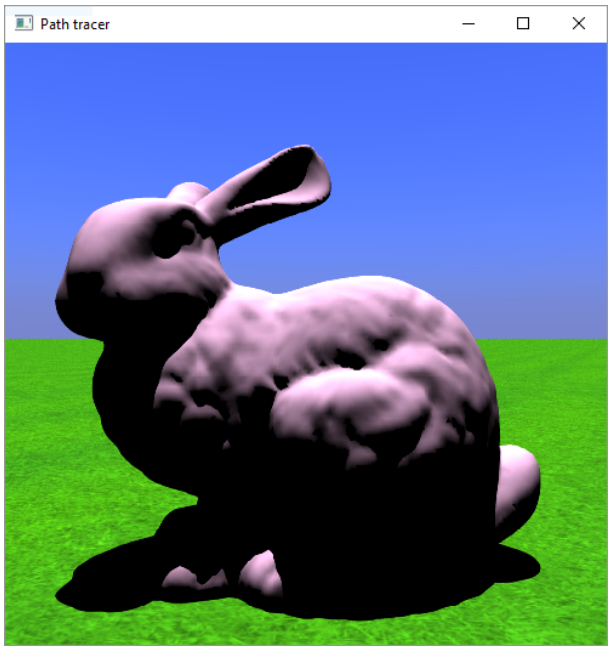
2

WORKSHEET 2 DELIVERABLES

We've implemented the sun light using the Preethams model to obtain RGB emission values. In order to go from the incident radiance given by the model to irradiance received at a surface (such that the total amount of energy emitted is correct), we use formula (1) in which we use the variable `_sun_color` (which comes from the Preetham model) as L_e, and the data from the slides for A_{sun} and r. A rendering of this can be seen in figure 7.

$$E = L_e \frac{A_{sun}}{4r^2} \tag{1}$$

Afterwards we implemented the moving of the sun during the day in the function `RenderEngine::set_time_of_day`. Renderings of this can be seen in figure 8, 9 and 10, where the sun moves from 7 in the morning to 17 in the afternoon.




```
Loading ../models/bunny.obj
Computing normals
No. of triangles: 69451
Loading ..\models\media.mpml
Background ambient: [ 0.8 0.9 1 ]
Building acceleration structure...(time: 0.58)
Building photon maps...
Particles in caustics map: 0 (photons shot: 1)
Particles in global map: 0 (photons shot: 1)
Building time: 0.005
Generating scene display list..........
Switched to shader number 1
Generating scene display list..........
Toggled textures on.
Generating scene display list..........
Rays per pixel: 4
Rays per pixel: 9
Rays per pixel: 16
Raytracing.......... - 4.532 secs
```

Figure 7: Stanford bunny with sunlight, before Preethams model implemented

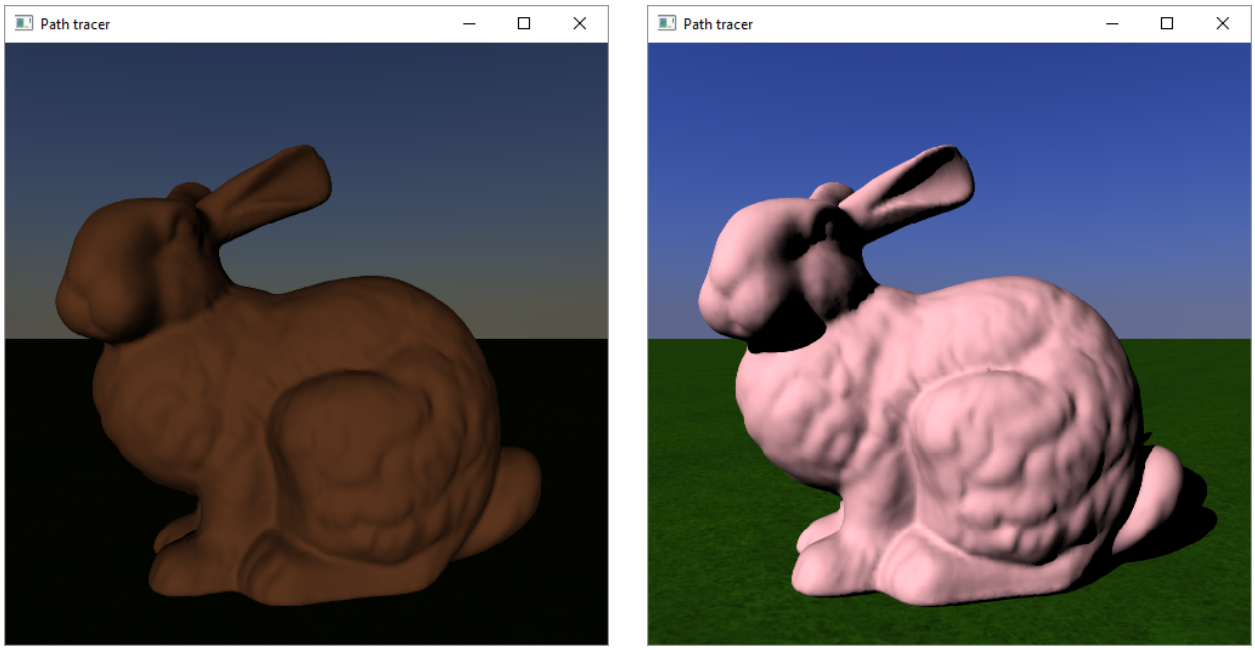


Figure 8: Stanford bunny with sunlight between 7 and 9

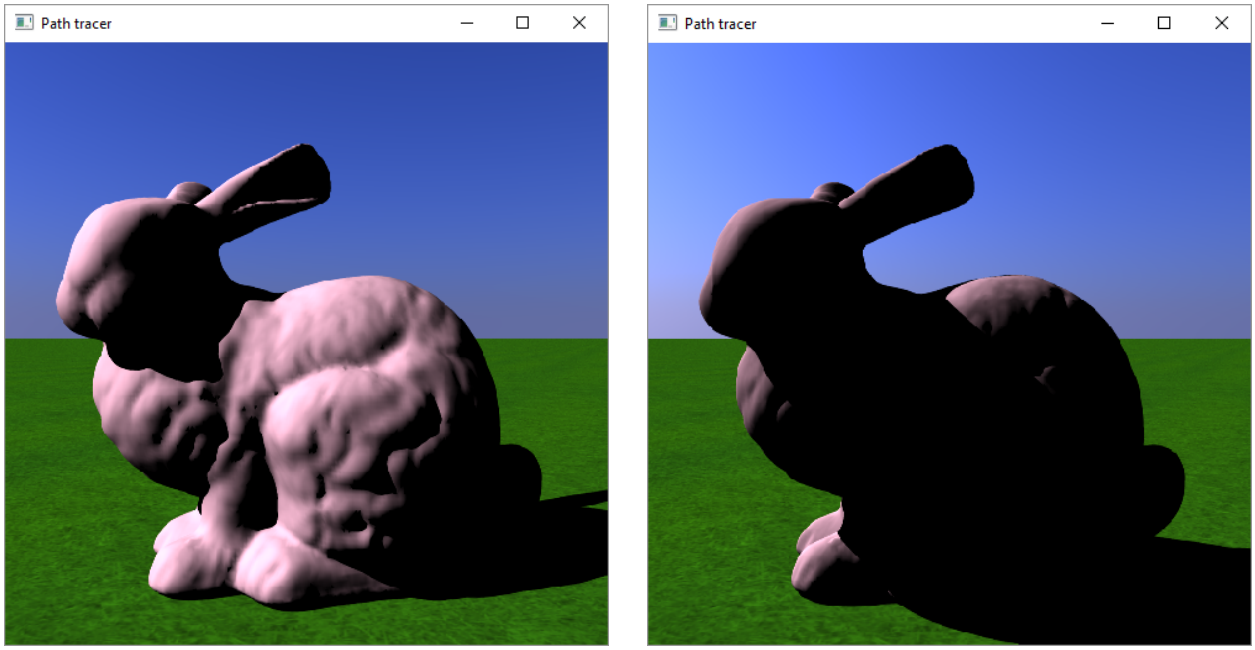


Figure 9: Stanford bunny with sunlight between 11 and 13

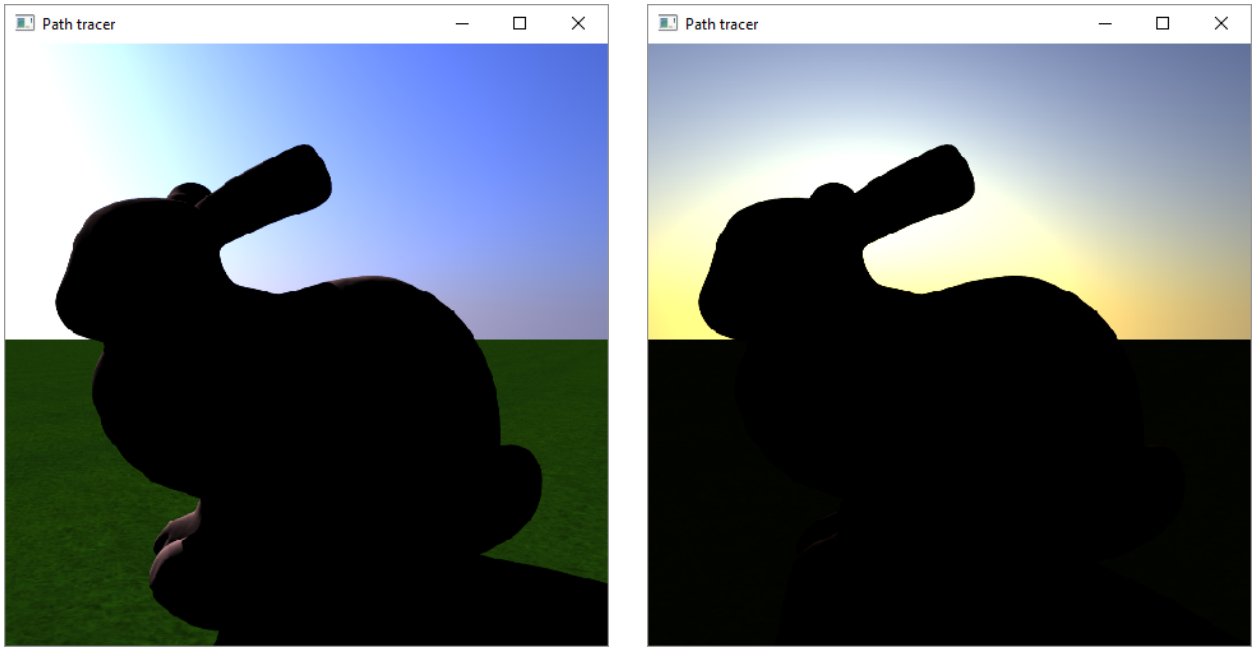


Figure 10: Stanford bunny with sunlight between 15 and 17

3

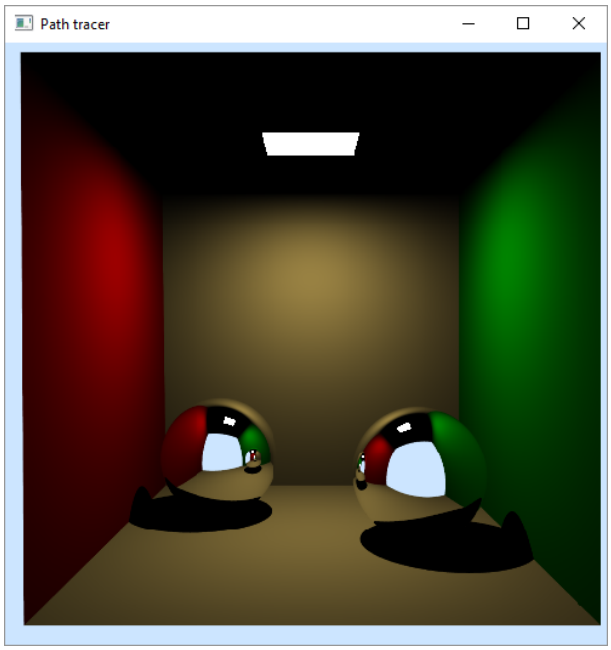
WORKSHEET 3 DELIVERABLES

We've implemented recursive ray tracing in the Mirror shader. A render of this can be seen in figure 11.

Afterwards we implemented a shader for transparent objects. A render of this can be seen in figure 12. We use the Fresnel equations to get the correct ratio of reflection and transmission.

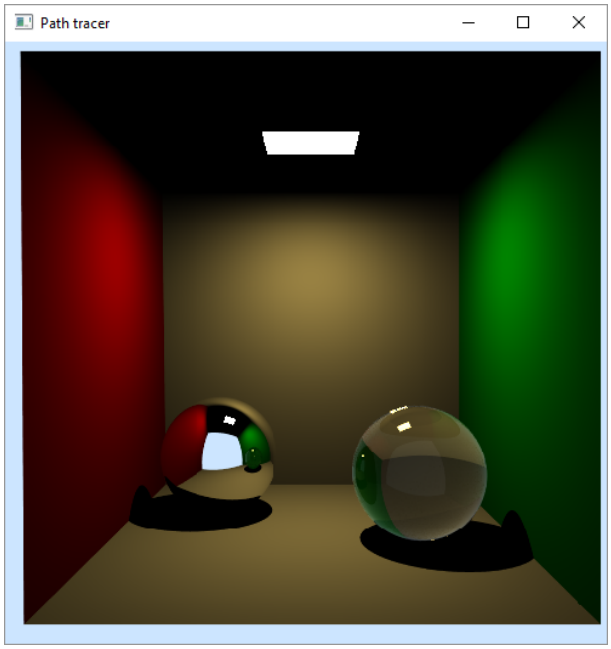
Finally we've implemented a metal shader. The color of metals stem from their complex index of refraction, as that means we can get the reflectance as a vector. This vector describes the amount of light reflected for each color component (in this case RGB). We use split shading for the first few rays and then switch to Russian roulette, using the mean of the reflectance vector as probability. A render of this can be seen in figure 13.

A render of a glass elephant and gold bunny can be seen in figure 14.



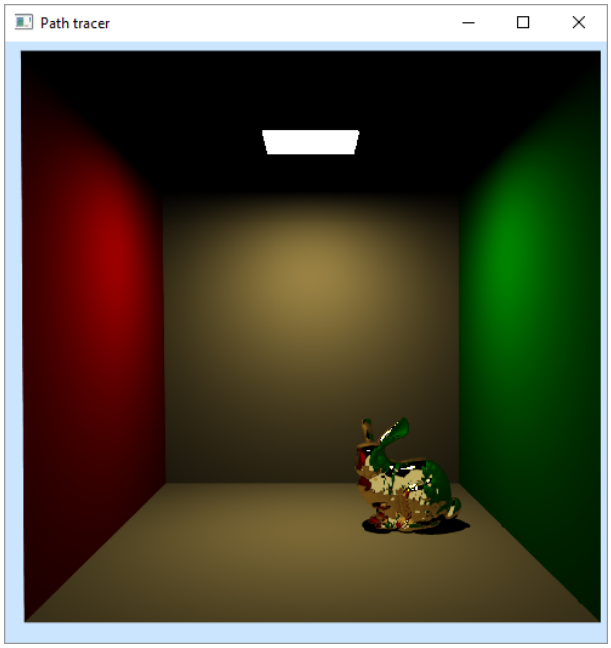

```
Loading ../models/CornellBox.obj
Computing normals
No. of triangles: 12
Loading ../models/CornellLeftSphere.obj
Computing normals
No. of triangles: 8073
Loading ../models/CornellRightSphere.obj
Computing normals
No. of triangles: 8073
Loading ..\models\media.mpml
Background ambient: [ 0.8 0.9 1 ]
Building acceleration structure...(time: 0.151)
Building photon maps...
Particles in caustics map: 0 (photons shot: 1)
Particles in global map: 0 (photons shot: 1)
Building time: 0.005
Generating scene display list...................
Rays per pixel: 4
Rays per pixel: 9
Rays per pixel: 16
Switched to shader number 1
Generating scene display list...................
Raytracing.......... - 11.76 secs
```

Figure 11: Two balls rendered using the mirror shader in the Cornell box



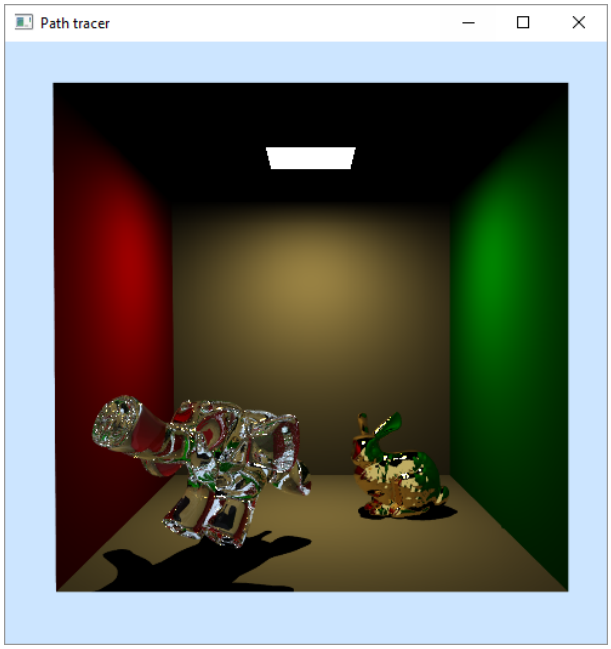

```
Loading ../models/CornellLeftSphere.obj
Computing normals
No. of triangles: 8073
Loading ../models/CornellRightSphere.obj
Computing normals
No. of triangles: 8073
Loading ../models/CornellBox.obj
Computing normals
No. of triangles: 12
Loading ..\models\media.mpml
Background ambient: [ 0.8 0.9 1 ]
Building acceleration structure...(time: 0.238)
Building photon maps...
Particles in caustics map: 0 (photons shot: 1)
Particles in global map: 0 (photons shot: 1)
Building time: 0.017
Generating scene display list...................
Rays per pixel: 4
Rays per pixel: 9
Rays per pixel: 16
Switched to shader number 1
Generating scene display list...................
Raytracing.......... - 14.674 secs
```

Figure 12: Two balls rendered in the Cornell box. Left: Mirror shader. Right: Transparent shader.




```
Loading ../models/bunny.obj
Computing normals
No. of triangles: 69451
Loading ../models/CornellBox.obj
Computing normals
No. of triangles: 12
Loading ..\models\media.mpml
Background ambient: [ 0.8 0.9 1 ]
Building acceleration structure...(time: 0.334)
Building photon maps...
Particles in caustics map: 0 (photons shot: 1)
Particles in global map: 0 (photons shot: 1)
Building time: 0.007
Generating scene display list..........
Rays per pixel: 4
Rays per pixel: 9
Rays per pixel: 16
Switched to shader number 1
Generating scene display list..........
Raytracing.......... - 12.344 secs
```

Figure 13: The Stanford bunny rendered using the metal shader with complex IOR values for gold in the Cornell box.




```
Loading ../models/bunny.obj
Computing normals
No. of triangles: 69451
Loading ../models/justElephant.obj
No. of triangles: 12064
Loading ../models/CornellBox.obj
Computing normals
No. of triangles: 12
Loading ..\models\media.mpml
Background ambient: [ 0.8 0.9 1 ]
Building acceleration structure...(time: 0.452)
Building photon maps...
Particles in caustics map: 0 (photons shot: 1)
Particles in global map: 0 (photons shot: 1)
Building time: 0.01
Generating scene display list...................
Rays per pixel: 4
Rays per pixel: 9
Rays per pixel: 16
Switched to shader number 1
Generating scene display list...................
Raytracing.......... - 18.62 secs
```

Figure 14: An elephant (transparent shader) and the Stanford bunny (metal shader with gold IOR) rendered in the Cornell box.

Listing 5: Transparent.cpp

```
Vec3f Transparent::shade(Ray& r, bool emit) const
{
    if (r.trace_depth < splits) {
        return split_shade(r, emit);
    }
    else if (r.trace_depth < max_depth) {
        Ray out;
        double R;
        tracer->trace_refracted(r, out, R);

        float xi = mt_random();

        if (xi < R) {
            tracer->trace_reflected(r, out);
        }

        return shade_new_ray(out);
    }

    return Vec3f(0.0f);
}

Vec3f Transparent::split_shade(Ray& r, bool emit) const
{

    Ray reflected, refracted;
    double R;

    tracer->trace_reflected(r, reflected);
    tracer->trace_refracted(r, refracted, R);

    return shade_new_ray(reflected) * R + (1 - R) *
        shade_new_ray(refracted);
}
```

Listing 6: `Metal.cpp`

```
Vec3f Metal::shade(Ray& r, bool emit) const
{
    Vec3f result(0.0f);

    Ray reflected;

    Vec3f R;
    tracer->trace_reflected(r, reflected, R);
    float probability = (R[0] + R[1] + R[2]) / 3.0f;

    if (r.trace_depth < splits) {
        return R * shade_new_ray(reflected);
    }
    else if (r.trace_depth < max_depth) {
        double xi = mt_random();
        if (xi < probability) {
            return (R * shade_new_ray(reflected)) /
                probability;
        }
    }

    return result;
}
```

4

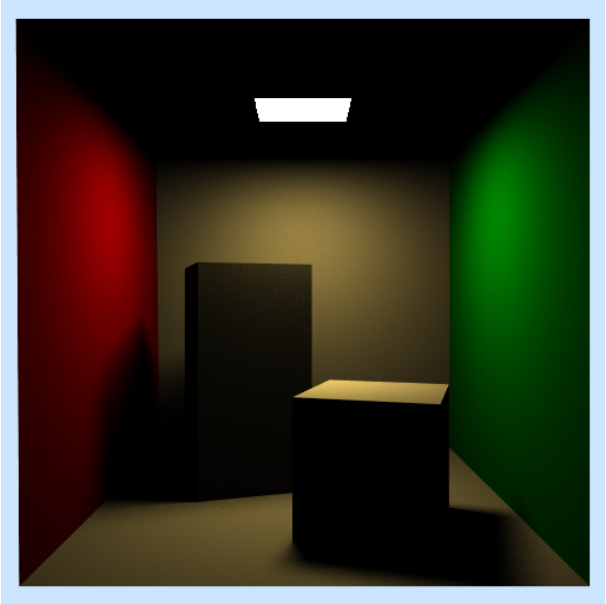
WORKSHEET 4 DELIVERABLES

We have updated the simplified area light implementation to sample a random position on the surface of the area light, so that we can apply Monte Carlo integration in the Lambertian shader and achieve soft shadowing. Our Lambertian shader already did this, but it previously had no effect because the simplified area light only used 1 sample. A rendering of the Cornell box where the blocks casts soft shadows can be seen in figure 15.

We have implemented an ambient occlusion shader. A rendering of the Stanford bunny using this shader and the sky and sun environment can be seen in figure 16.

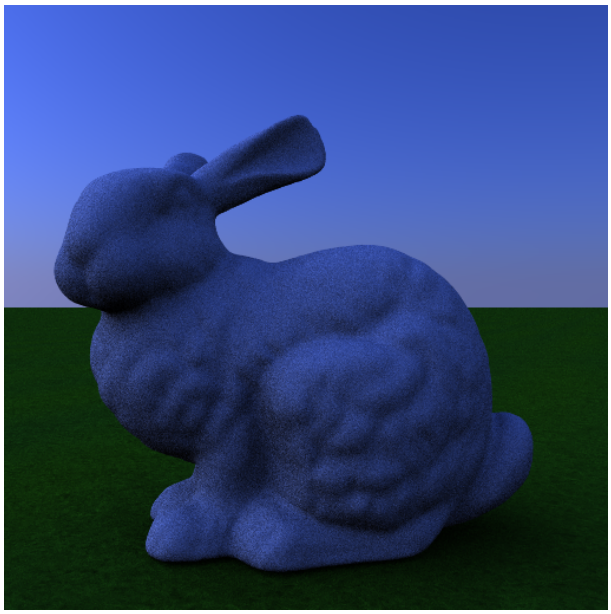
The direct illumination from the Lambertian shader has been added to this shader. Renderings of this can be seen in figure 17, 18 and 19, where the sun moves from 7 in the morning to 17 in the afternoon.

A rendering of the elephant and the Stanford bunny in the Cornell box using both ambient occlusion and direct illumination can be seen in figures 20 and 21. Both use 16 rays per pixel. The former uses 5 occlusion rays and 4 light samples per pixel. The latter uses 25 occlusion rays and 16 light samples per pixel, which causes a rather big increase in both quality and render time.



```
Loading ../models/CornellBox.obj
Computing normals
No. of triangles: 12
Loading ../models/CornellBlocks.obj
Computing normals
No. of triangles: 24
Loading ..\models\media.mpml
Background ambient: [ 0.8 0.9 1 ]
Building acceleration structure...(time: 0.009)
Building photon maps...
Particles in caustics map: 0 (photons shot: 1)
Particles in global map: 0 (photons shot: 1)
Building time: 0.01
Generating scene display list
Rays per pixel: 4
Rays per pixel: 9
Rays per pixel: 16
Switched to shader number 1
Generating scene display list
Raytracing.......... - 36.958 secs
Rendered image stored in cornellblocks.png.
```

Figure 15: Cornell Box with area light and soft shadows. 16 Rays per pixel and 16 light samples per pixel



```
Loading ../models/bunny.obj
Computing normals
No. of triangles: 69451
Loading ..\models\media.mpml
Background ambient: [ 0.8 0.9 1 ]
Building acceleration structure...(time: 0.647)
Building photon maps...
Particles in caustics map: 0 (photons shot: 1)
Particles in global map: 0 (photons shot: 1)
Building time: 0.006
Generating scene display list..........
Switched to shader number 2
Generating scene display list..........
Raytracing.......... - 1.228 secs
Rays per pixel: 4
Rays per pixel: 9
Rays per pixel: 16
Raytracing.......... - 17.335 secs
Toggled textures on.
Generating scene display list..........
Raytracing.......... - 17.459 secs
Rendered image stored in bunny.png.
```

Figure 16: Stanford Bunny with Ambient Occlusion, set in the skylight scene. 16 Rays per pixel

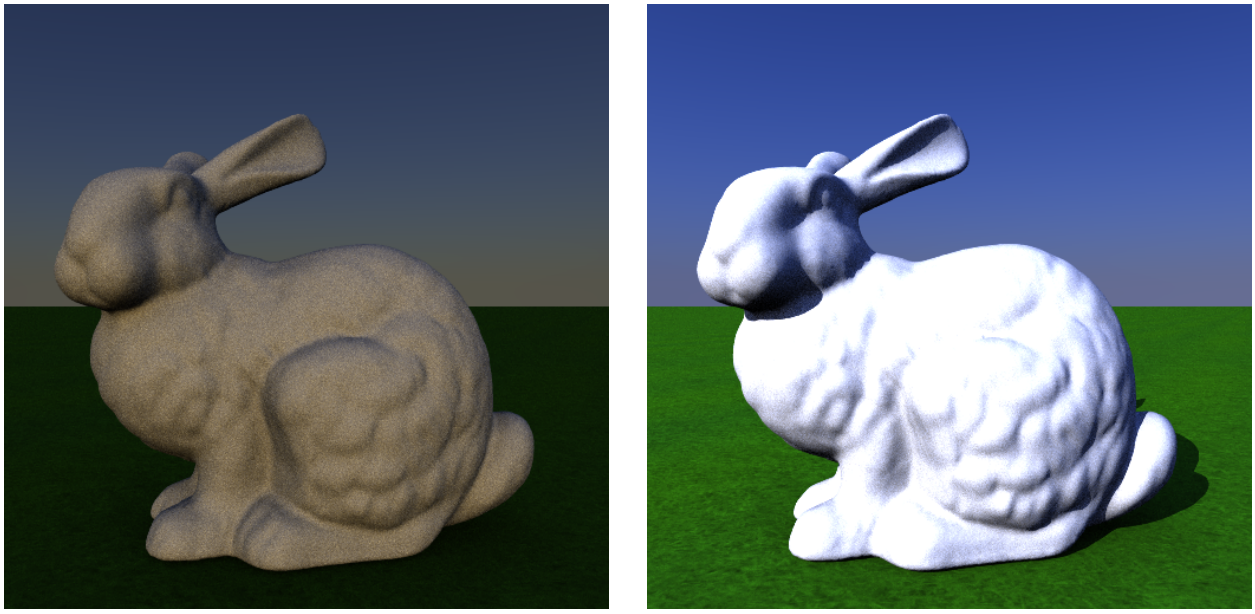


Figure 17: Stanford bunny with sunlight between 7 and 9. Combined skylight and directional light.

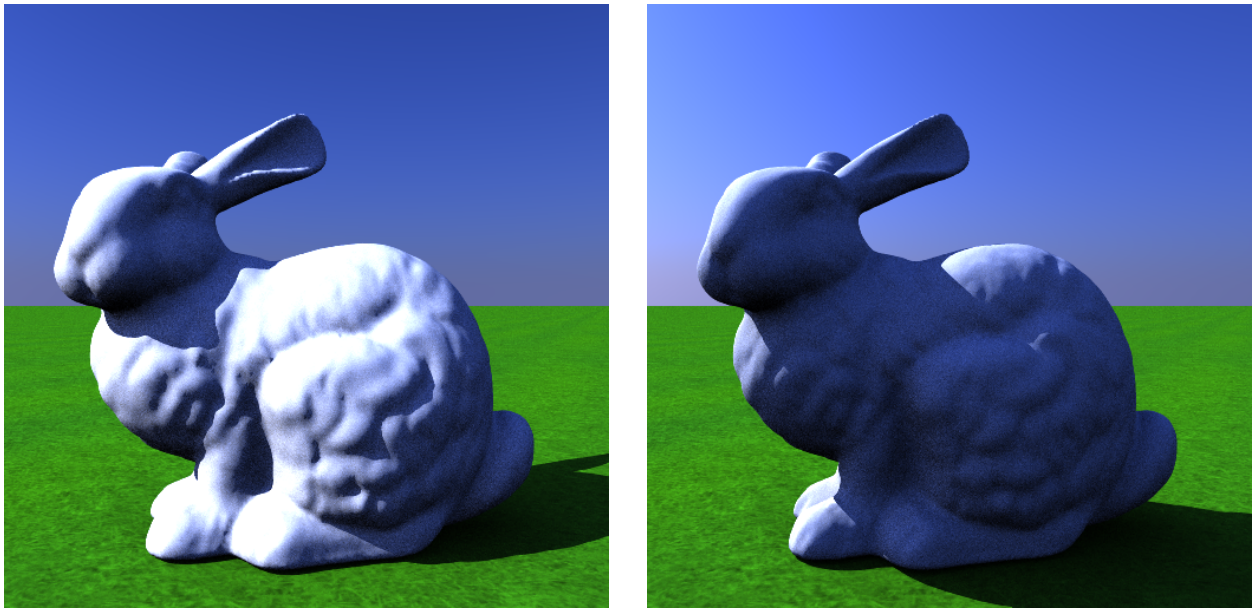


Figure 18: Stanford bunny with sunlight between 11 and 13. Combined skylight and directional light.

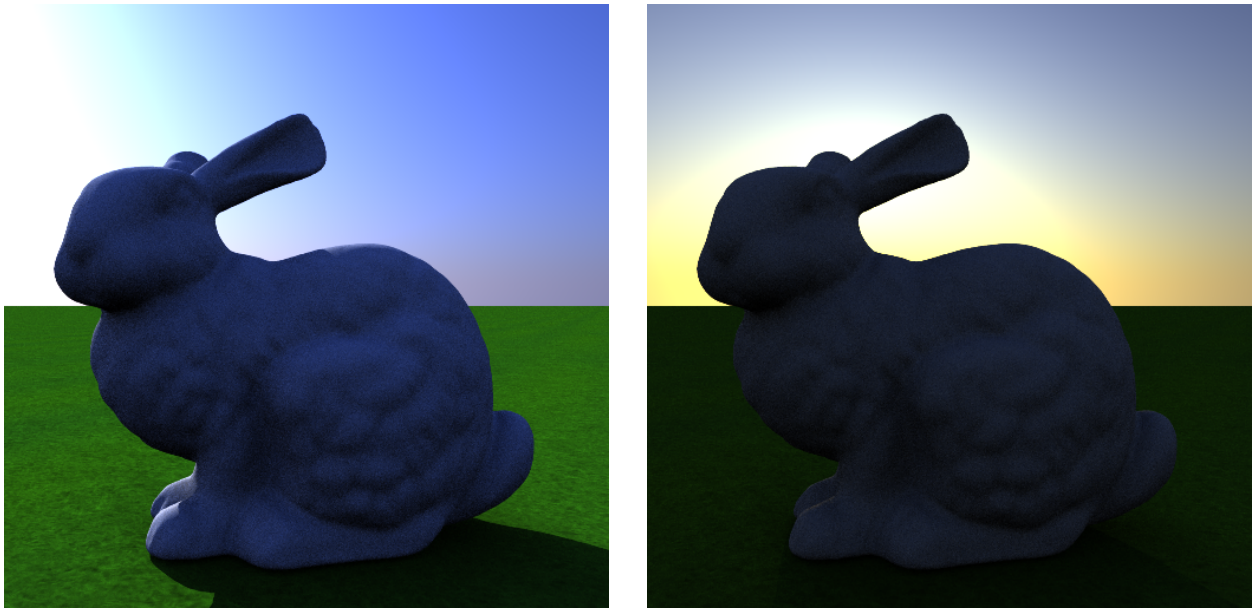
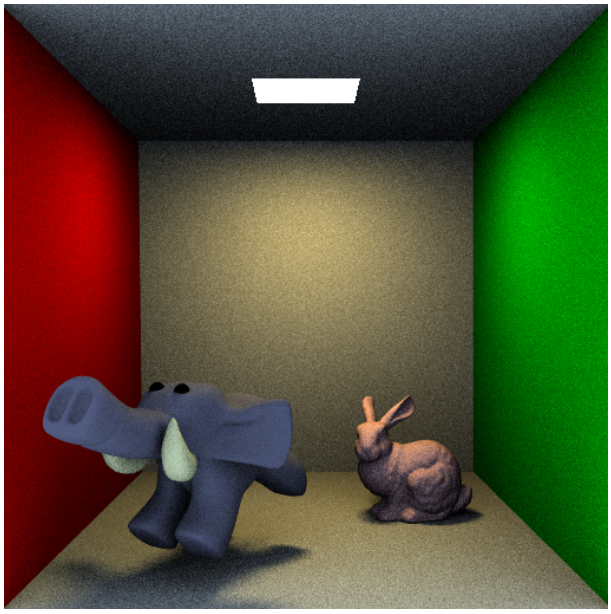
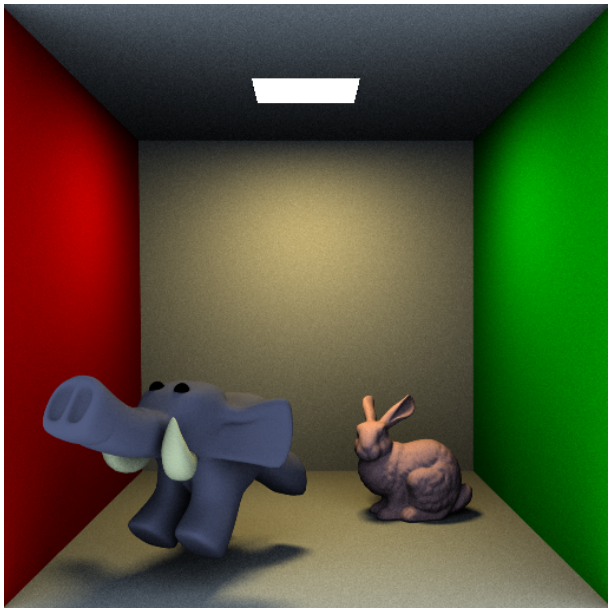


Figure 19: Stanford bunny with sunlight between 15 and 17. Combined skylight and directional light.



```
Loading ../models/justElephant.obj
No. of triangles: 12064
Loading ../models/bunny.obj
Computing normals
No. of triangles: 69451
Loading ../models/CornellBox.obj
Computing normals
No. of triangles: 12
Loading ..\models\media.mpml
Background ambient: [ 0.8 0.9 1 ]
Building acceleration structure...(time: 0.44)
Building photon maps...
Particles in caustics map: 0 (photons shot: 1)
Particles in global map: 0 (photons shot: 1)
Building time: 0.005
Generating scene display list...................
Rays per pixel: 4
Rays per pixel: 9
Rays per pixel: 16
Switched to shader number 2
Generating scene display list...................
Raytracing.......... - 38.556 secs
Rendered image stored in cornellbox.png.
```

Figure 20: Cornell Box with Standford Bunny and justElephant. 16 Rays per pixel, 5 occlusion rays per pixel and 4 light samples per pixel



```
Loading ../models/justElephant.obj
No. of triangles: 12064
Loading ../models/bunny.obj
Computing normals
No. of triangles: 69451
Loading ../models/CornellBox.obj
Computing normals
No. of triangles: 12
Loading ..\models\media.mpml
Background ambient: [ 0.8 0.9 1 ]
Building acceleration structure...(time: 0.557)
Building photon maps...
Particles in caustics map: 0 (photons shot: 1)
Particles in global map: 0 (photons shot: 1)
Building time: 0.011
Generating scene display list.....................
Switched to shader number 2
Generating scene display list.....................
Raytracing.......... - 10.397 secs
Rays per pixel: 4
Rays per pixel: 9
Rays per pixel: 16
Raytracing.......... - 175.655 secs
Rendered image stored in cornellbox.png.
```

Figure 21: Cornell Box with Standford Bunny and justElephant. 16 Rays per pixel, 25 occlusion rays per pixel and 16 light samples per pixel

Listing 7: AreaLight.cpp

```
bool AreaLight::sample(const Vec3f& pos, Vec3f& dir, Vec3f&
    L) const
{
    // Get geometry info
    const IndexedFaceSet& geometry = mesh->geometry;
    const IndexedFaceSet& normals = mesh->normals;
    const float no_of_faces = geometry.no_faces();
    const float no_of_vertices = geometry.no_vertices();

    auto triangle_index = static_cast<int>(
        mt_random_half_open() * no_of_faces);
    auto xi1 = mt_random();
    auto xi2 = mt_random();

    auto u = 1 - sqrt(xi1);
    auto v = (1 - xi2) * sqrt(xi1);
    auto w = xi2 * sqrt(xi1);

    auto face = geometry.face(triangle_index);

    // Triangle vertices
    auto x0 = geometry.vertex(face[0]);
    auto x1 = geometry.vertex(face[1]);
    auto x2 = geometry.vertex(face[2]);

    // Triangle edges
    auto e1 = x1 - x0;
    auto e2 = x2 - x0;

    auto L_e = get_emission(triangle_index);
    auto A_i = 0.5f * cross(e1, e2).length();

    auto sample_pos = u * x0 + v * x1 + w * x2;
    auto sample_normal = (u * normals.vertex(face[0]) + v *
         normals.vertex(face[1]) + w * normals.vertex(face
        [2]));
    sample_normal.normalize();

    auto d = sample_pos - pos;
    dir = normalize(d);

    float cos_theta = dot(-dir, sample_normal);
    float r_sqr = d.length() * d.length();

    L = no_of_faces * L_e * A_i * cos_theta / r_sqr;

    if (shadows) {
        Ray shadow_ray(pos, dir, 1.0e-4f, d.length() - 1.0e
            -4f);
        tracer->trace(shadow_ray);
        return !shadow_ray.has_hit;
    }

    return true;
}
```

Listing 8: `Ambient.cpp`

```
Vec3f Ambient::shade(Ray& r, bool emit) const
{
    Vec3f ambient(0.0f);

    for (size_t i = 0; i < samples; ++i) {
        Ray r_out;
        if (!tracer->trace_cosine_weighted(r, r_out)) {
            ambient += tracer->get_background(r_out.
                direction);
        }
    }

    ambient /= samples;
    return ambient*get_diffuse(r) + Lambertian::shade(r,
        emit);
}
```

5

WORKSHEET 5 DELIVERABLES

We've implemented the Monte Carlo version of the Lambertian shader. It uses split shading for the first diffuse surface and then Russian roulette afterwards. We've also implemented a version that only uses Russian roulette. The two different versions can be seen side by side in figure 22, 23, 24 and 25. Split + Russian Roulette to the right and only russian roulette to the left.

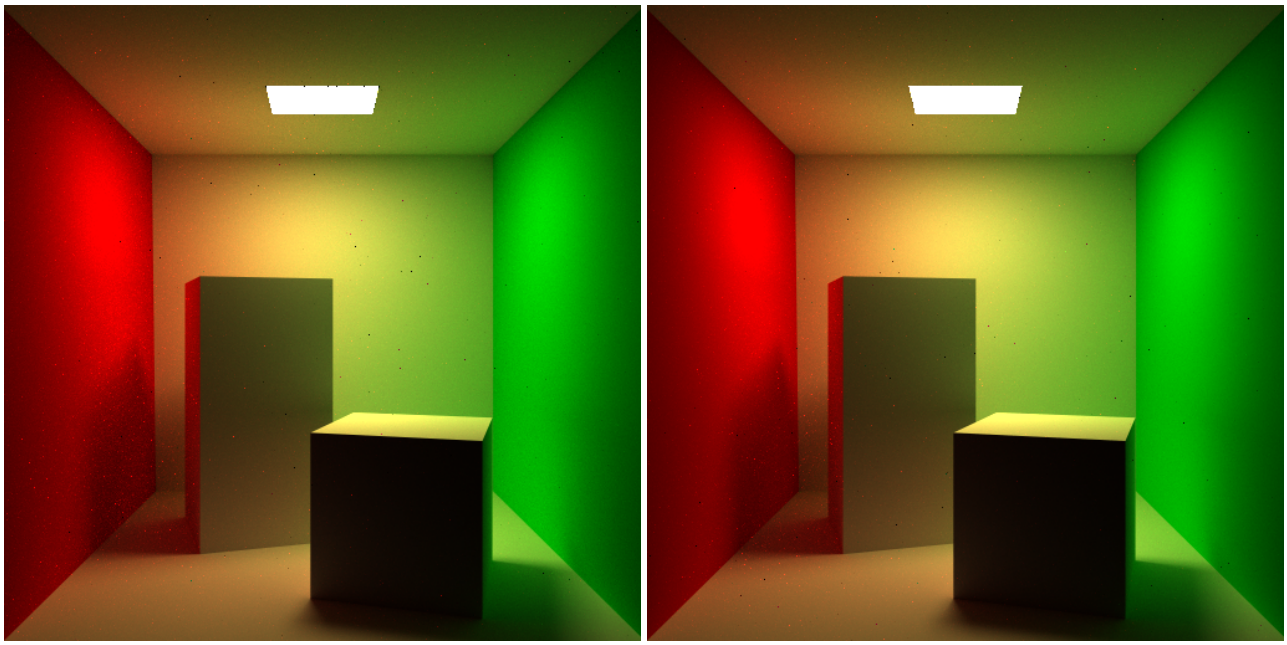
We have the possibility to use 2 different probabilities when deciding whether a path should terminate or not: Luminance and the average of the 3 color components. We've found that the luminance method does not work too well for the Cornell box. The green side looks very good, but the red side looks worse. Comparison can be seen between figure 22, which use luminance, and 23 which use the average of the colour.

We've compared using split shading for the first diffuse surface with not using split shading at all. For the render times we've used, the result is that the version without split has noticeably more variance. But it should be noted that in the renders, the caustics generally haven't gotten enough samples to be perfect. Over time, the version using only Russian roulette might be more efficient, as the caustics require so many samples that the split shading will cause way too more samples to be taken than is actually needed for the diffuse surfaces.

Indirect light paths can be seen as blue coloured lines in figure 26, along with the direct light paths as pink lines.

The blue ray to the left of the rabbit and to the right of the chrome sphere, where the light which bounce of a diffuse wall are reflected off the specular material and presents itself colouring the floor the same colour as the wall the light bounced off. Red for the rabbit and Green for the sphere.

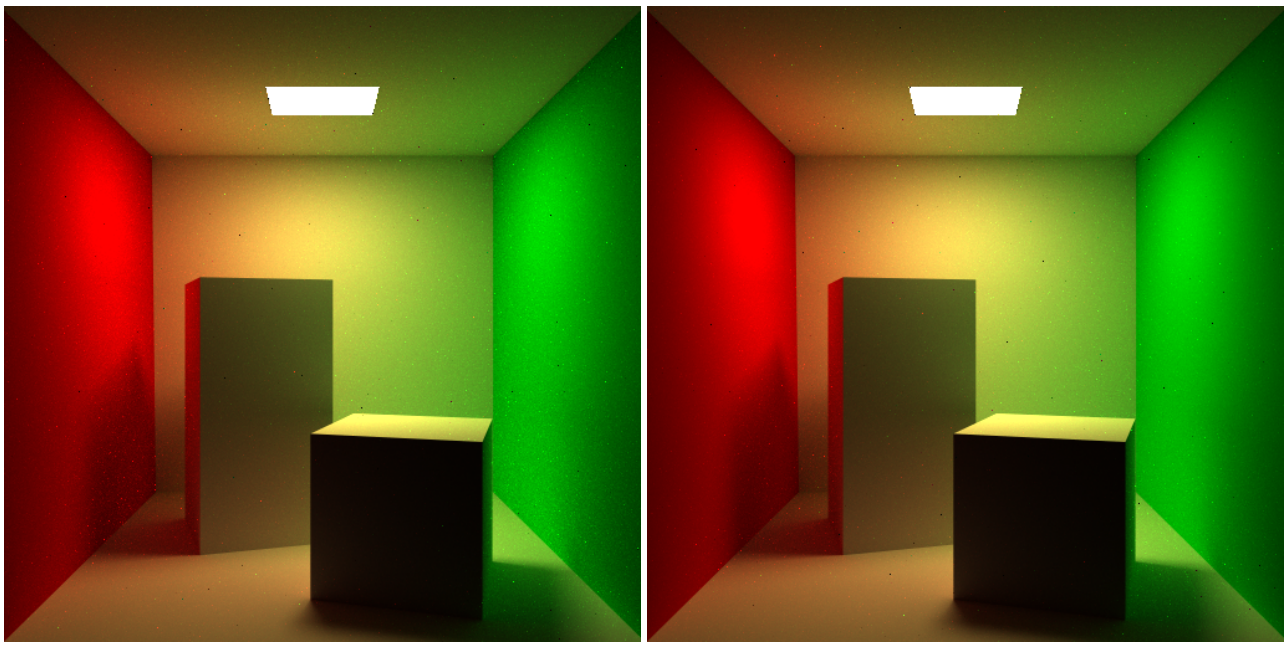
The pink ray presents itself as caustics from the glass sphere and to the ground. There is also the ray which goes through the elephant and to the ground. Lastly there's the one which reflects off the mirror ball, through the glass ball and then onto the green wall. It is clear that the intensity of that light is quite a lot lower than the direct caustic seen on the ground.



(a) 6,157 frames, 7,220.97 seconds, 4 threads

(b) 1,219 frames, 7,116.53 seconds, 4 threads

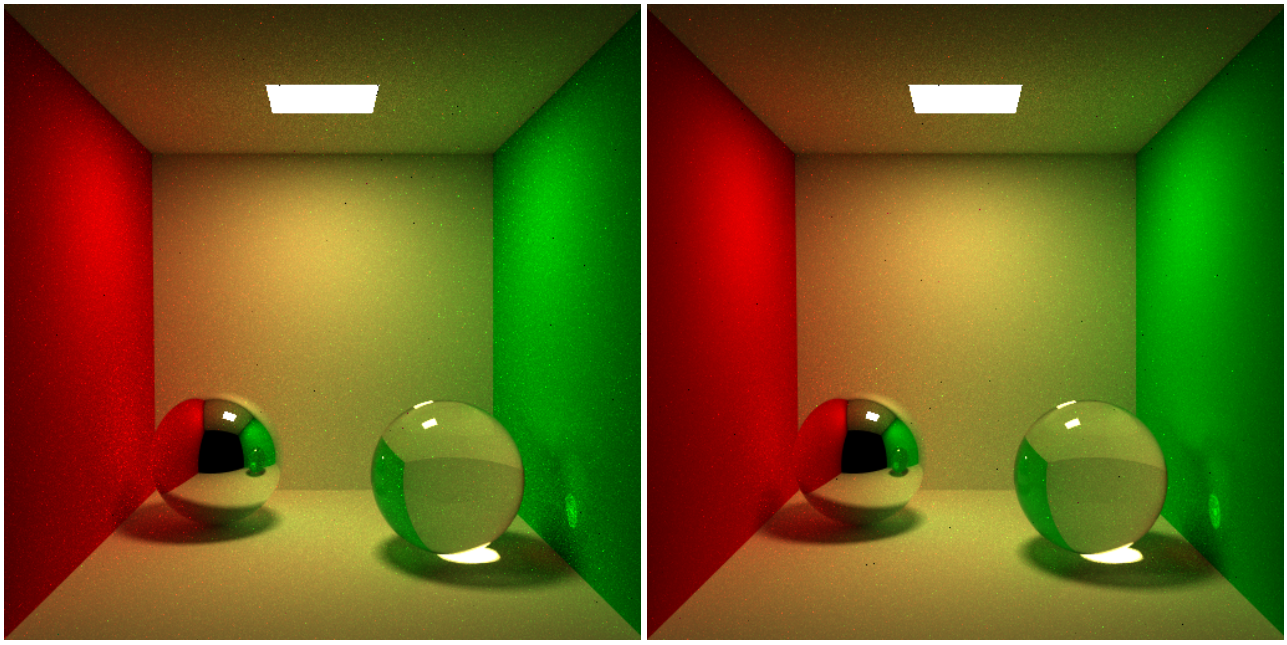
Figure 22: Left: Only Russian Roulette, Right: Splitting on first hit. Cornell Box with blocks, luminance used in probability



(a) 6,005 frames, 5,490.26 seconds, 8 threads

(b) 1,201 frames, 5,136.93 seconds, 8 threads

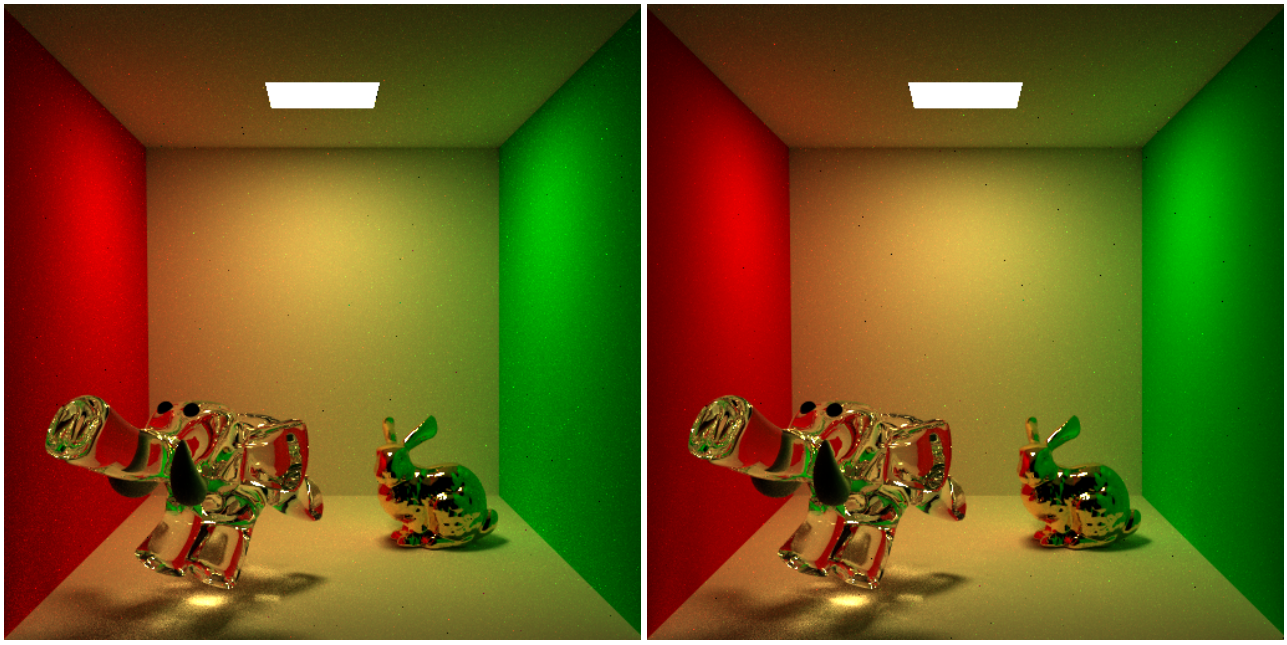
Figure 23: Left: Only Russian Roulette, Right: Splitting on first hit. Cornell Box with blocks, average colour used in probability



(a) 3,373 frames, 4,299.41 seconds, 16 threads

(b) 653 frames, 3,623.51 seconds, 16 threads

Figure 24: Left: Only Russian Roulette, Right: Splitting on first hit. Cornell Box with spheres, one glass, one silver. Average colour used in probability



(a) 3,800 frames, 5,679.78 seconds, 8 threads

(b) 757 frames, 4,560.19 seconds, 8 threads

Figure 25: Left: Only Russian Roulette, Right: Splitting on first hit. Cornell Box with Stanford Bunny and justElephant. Gold Bunny, glass elephant. Average colour used in probability

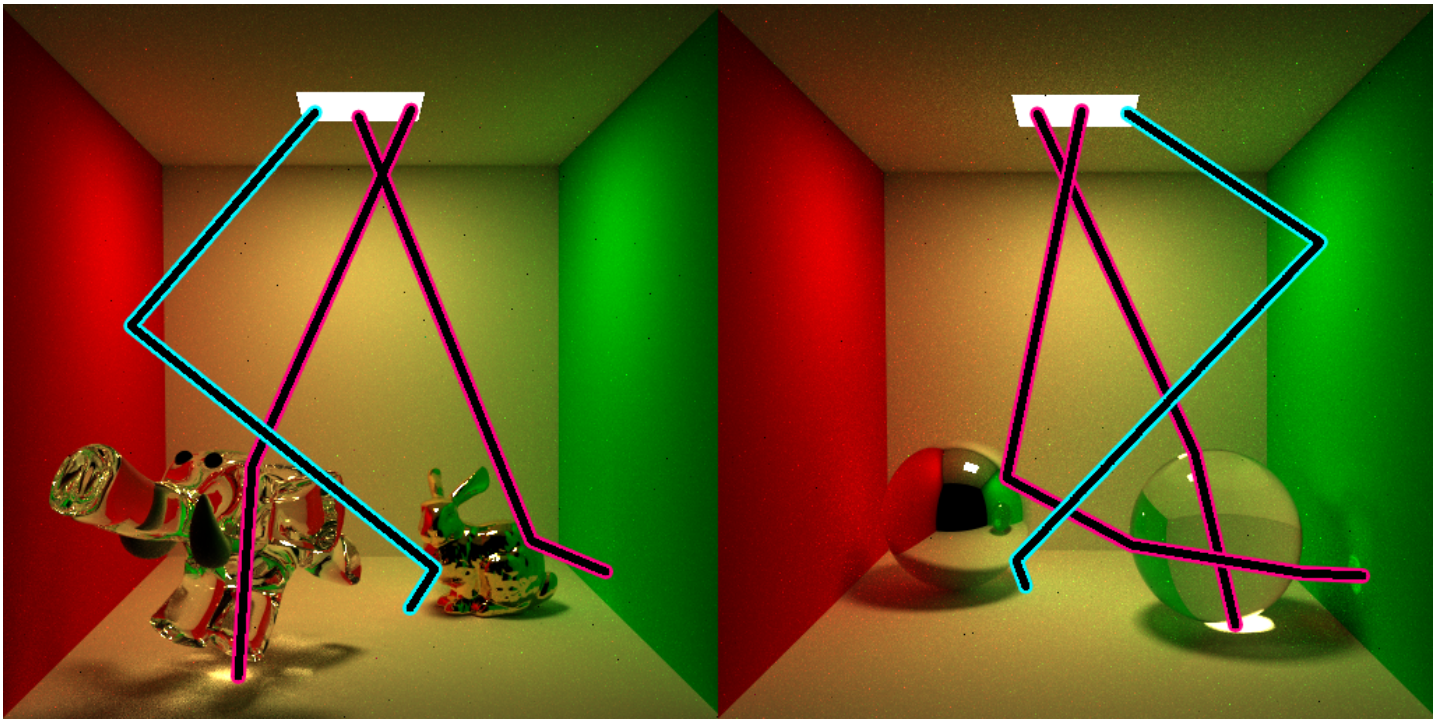


Figure 26: Pink lines: The path the direct light takes from emitter to the diffuse material. Blue lines: The path the indirect light takes from the emitter, through specular material, to the diffuse material.

Listing 9: Implementation of Monte Carlo Lambertian shader

```
Vec3f MCLambertian::shade(Ray& r, bool emit) const
{
    Vec3f result = Lambertian::shade(r, emit);

    if (!r.did_hit_diffuse) {
        result += split_shade(r, emit);
    }
    else {
        result += russian_roulette(r, emit);
    }

    return result;
}

Vec3f MCLambertian::split_shade(Ray& r, bool emit) const
{
    Vec3f rho_d = get_diffuse(r);
    Vec3f result(0.0f);

    for (size_t i = 0; i < samples; ++i) {
        Ray r_out;
        tracer->trace_cosine_weighted(r, r_out);
        result += shade_new_ray(r_out);
    }

    return rho_d * result / samples;
}

Vec3f MCLambertian::russian_roulette(Ray & r_in, bool emit)
    const
{
    Vec3f rho_d = get_diffuse(r_in);
    double luminance = get_luminance(rho_d);
    double avg = (rho_d[0] + rho_d[1] + rho_d[2]) / 3.0;
    double xi = mt_random();

    if (xi < avg) {
        Ray r_out;
        tracer->trace_cosine_weighted(r_in, r_out);
        return rho_d * shade_new_ray(r_out) / avg;
    }

    return Vec3f(0.0f);
}
```

6

WORKSHEET 6 DELIVERABLES

We have implemented photon mapping for Lambertian surfaces (see figure 28 and 29) and for caustics (see figure 30 and 31). Photon mapping is biased because it only calculates an estimate of the irradiance estimate (by using the photon map). Thus a rather big term of the rendering equation is not precise. Especially note the edges of the Cornell box in figure 27, which clearly shows that the photon map is imprecise. In this case it is because of the density estimation not being good enough.

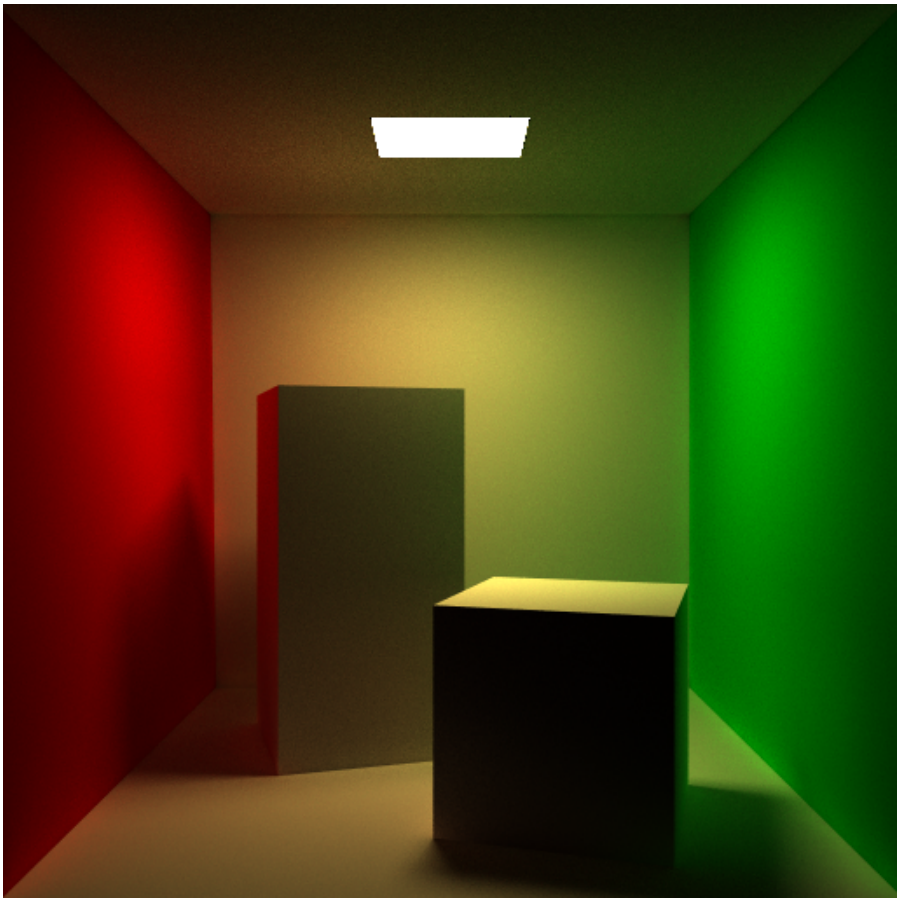


Figure 27: Cornell box and Cornell blocks rendered using photon mapping with final gathering, 100,000 photons and 231 frames taking a total of 8,689.94 seconds

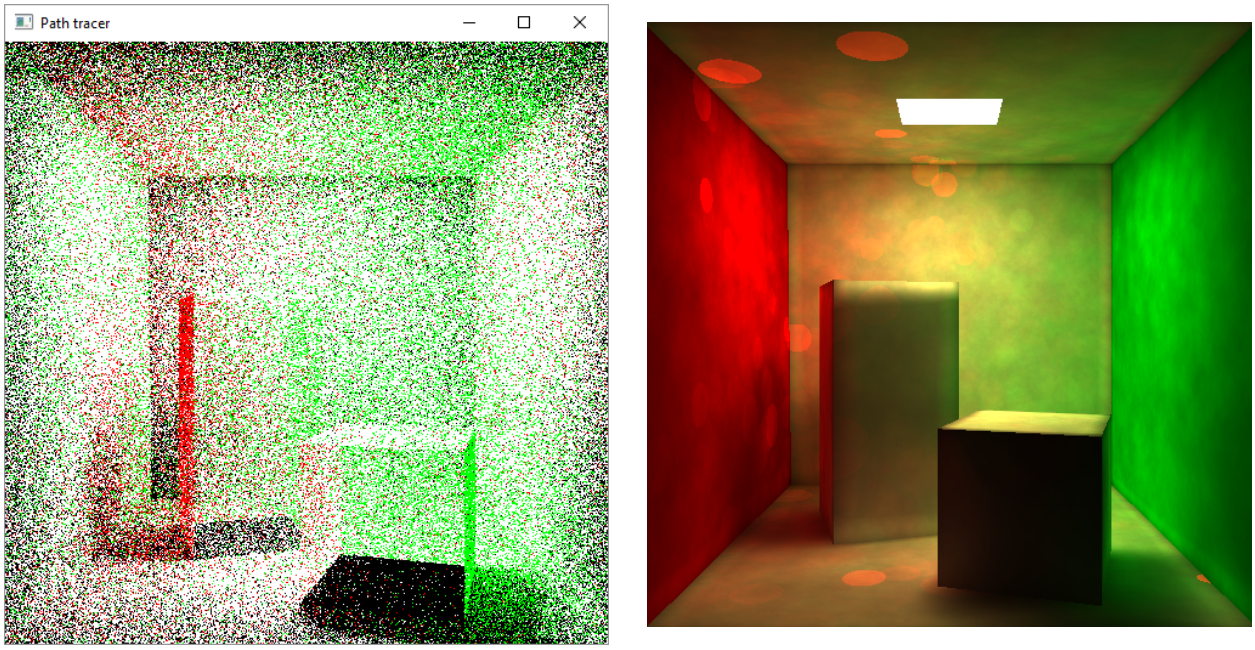
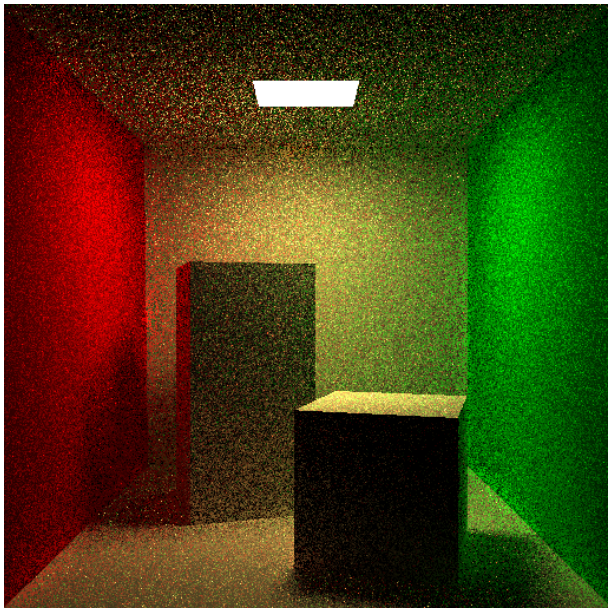


Figure 28: Photon map and estimated radiance



```
Loading ../models/CornellBlocks.obj
Computing normals
No. of triangles: 24
Loading ../models/CornellBox.obj
Computing normals
No. of triangles: 12
Loading ..\models\media.mpml
Background ambient: [ 0 0 0 ]
Building acceleration structure...(time: 0.009)
Building photon maps...
Particles in caustics map: 0 (photons shot: 1)
Particles in global map: 1000000 (photons shot:
    470000)
Building time: 3.063
Generating scene display list
Switched to shader number 5
Toggled final gathering off
Raytracing.......... - 14.674 secs
Rendered image stored in cornellbox.png.
Toggled final gathering on
Raytracing.......... - 98.953 secs
Rendered image stored in cornellbox.png.
```

Figure 29: Combined Photon map and lambertian shader with final gatherer

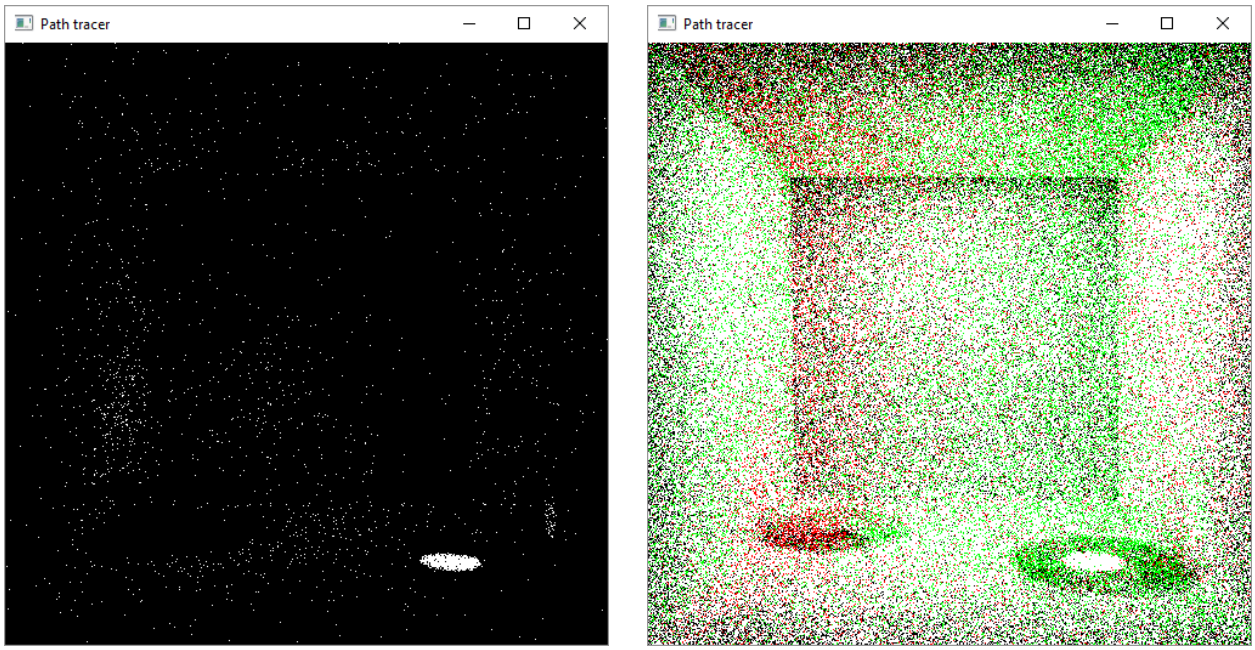


Figure 30: Photon map of caustics and combined caustics and lambertian

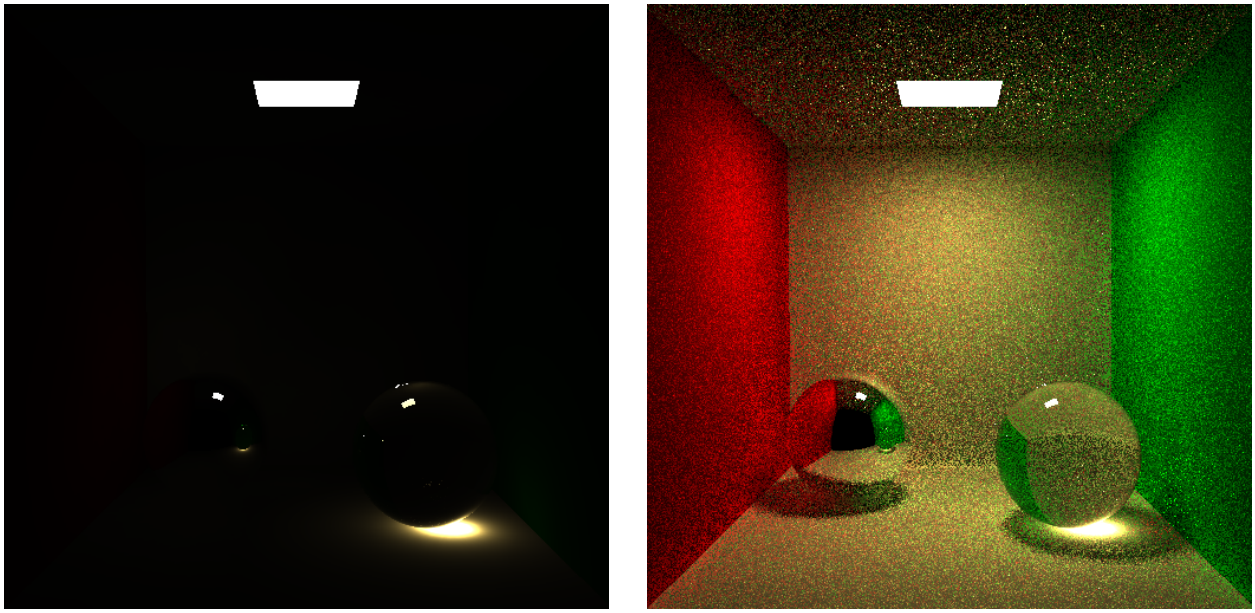


Figure 31: Combined Photon map, both lambertian and caustics with final gatherer

```
Loading ../models/CornellRightSphere.obj
Computing normals
No. of triangles: 8073
Loading ../models/CornellLeftSphere.obj
Computing normals
No. of triangles: 8073
Loading ../models/CornellBox.obj
Computing normals
No. of triangles: 12
Loading ..\models\media.mpml
Background ambient: [ 0 0 0 ]
Building acceleration structure...(time: 0.187)
Building photon maps...
Unable to store enough particles.
Particles in caustics map: 3058 (photons shot: 50000)
Particles in global map: 889516 (photons shot: 500000)
Building time: 2.855
Generating scene display list.....................
Switched to shader number 4
Toggled final gathering off
Raytracing.......... - 4.001 secs
Rendered image stored in cornellbox.png.
Switched to shader number 5
Toggled final gathering on
Raytracing.......... - 140.428 secs
Rendered image stored in cornellbox.png.
```

Figure 32: Console output for the Cornell Box with spheres, with both lambertian photons and caustic photons

Listing 10: Area light implementation

```
bool AreaLight::emit(Ray& r, Vec3f& Phi) const
{
    // Get geometry info
    const IndexedFaceSet& geometry = mesh->geometry;
    const IndexedFaceSet& normals = mesh->normals;
    const float no_of_faces = geometry.no_faces();

    // Sample ray origin and direction
    const auto triangle_index = sample_triangle_index();
    Vec3f normal;
    sample_pos(triangle_index, r.origin, normal);
    r.direction = sample_cosine_weighted(normal);

    // Trace ray
    // If a surface was hit, compute Phi and return true

    if (tracer->trace(r)) {
        const auto L_e_triagle = get_emission(
            triangle_index);
        const auto A_triangle = get_area(triangle_index);

        Phi = L_e_triagle * no_of_faces * A_triangle * M_PI
            ;
        return true;
    }

    return false;
}
```

Listing 11: Helper functions for area light implementation

```
size_t AreaLight::sample_triangle_index() const
{
    const IndexedFaceSet& geometry = mesh->geometry;
    const float no_of_faces = geometry.no_faces();

    return static_cast<int>(mt_random_half_open() *
        no_of_faces);;
}

void AreaLight::sample_pos(size_t triangle_id, CGLA::Vec3f&
    pos, CGLA::Vec3f& normal) const
{
    const IndexedFaceSet& geometry = mesh->geometry;
    const IndexedFaceSet& normals = mesh->normals;

    const auto xi1 = mt_random();
    const auto xi2 = mt_random();

    const auto u = 1 - sqrt(xi1);
    const auto v = (1 - xi2) * sqrt(xi1);
    const auto w = xi2 * sqrt(xi1);

    const auto face = geometry.face(triangle_id);

    pos = u * geometry.vertex(face[0]) + v * geometry.
        vertex(face[1]) + w * geometry.vertex(face[2]);
    normal = (u * normals.vertex(face[0]) + v * normals.
        vertex(face[1]) + w * normals.vertex(face[2]));
    normal.normalize();
}

float AreaLight::get_area(size_t triangle_id) const
{
    const IndexedFaceSet& geometry = mesh->geometry;
    const auto face = geometry.face(triangle_id);

    const auto x0 = geometry.vertex(face[0]);
    const auto x1 = geometry.vertex(face[1]);
    const auto x2 = geometry.vertex(face[2]);

    const auto e1 = x1 - x0;
    const auto e2 = x2 - x0;

    return 0.5f * cross(e1, e2).length();
}
```

Listing 12: Implementation of Photon Mapping Lambertian shader

```
Vec3f PhotonLambertian::shade(Ray& r, bool emit) const
{
    if (gather && !r.did_hit_diffuse) {
        return split_shade(r, emit);
    }

    Vec3f rho_d = get_diffuse(r);
    Vec3f radiance_estimate = rho_d * M_1_PIf * tracer->
        global_irradiance(r, max_dist, photons);

    return radiance_estimate + Emission::shade(r, !gather);
}

Vec3f PhotonLambertian::split_shade(Ray& r_in, bool emit)
    const
{
    Vec3f rho_d = get_diffuse(r_in);
    Vec3f result = Lambertian::shade(r_in, emit);

    if (rho_d[0] + rho_d[1] + rho_d[2] > 0.0) {
        Vec3f indirect(0.0f);
        Ray r_out;
        for (size_t i = 0; i < samples; i++) {
            tracer->trace_cosine_weighted(r_in, r_out);
            indirect += shade_new_ray(r_out);
        }

        indirect = rho_d * indirect / samples;
        result += indirect;

        // Add a radiance estimate from the caustics photon
            map
        result += caustics
            ? caustics->shade(r_in, false)
            : PhotonCaustics::shade(r_in, false);
    }

    return result;
}
```

Listing 13: Implementation of Photon Mapping Caustics shader

```
Vec3f PhotonCaustics::shade(Ray& r, bool emit) const
{
    Vec3f rho_d = get_diffuse(r);
    Vec3f radiance_estimate = rho_d * M_1_PIf * tracer->
        caustics_irradiance(r, max_dist, photons);

    return radiance_estimate + Emission::shade(r, emit);
}
```

7

WORKSHEET 7 DELIVERABLES

Why is dense flint glass better than crown glass for dispersion prisms?

It is better for dispersing the light rays, since the reflective index has a wider range of values of reflective index from low to high wavelengths compared to crown glass, at the same wavelengths. See figure 33.

We've implemented spectral rendering. In order to test this we've rendered glass objects which exhibit this behavior. Figures 34 and 35 shows a rendering of this using crown glass. The latter uses the improved density estimation. Figure 36 shows a rendering of this using dense flint glass. Unfortunately the improved density estimation also smoothens the caustics on the ground too much, making it lose it's sharp start.

We've improved the density estimation in our photon mapping. Renders without final gather can be seen in figure 37. The first improvement was to use an alternative kernel. This clearly gives the biggest improvement, although there is still some amount of light leaking. Next up we use a normal check to make sure that we only use photons where the normal at the position of the photon points in the same general direction as the normal for the point we're estimating. Now a lot of the light leaking disappears, but it also introduces a lot of "shadowing" at the edges. Our final improvement is to adjust the surface area according to the positions of the photons used. This removes even more light leaking and also the shadowing.

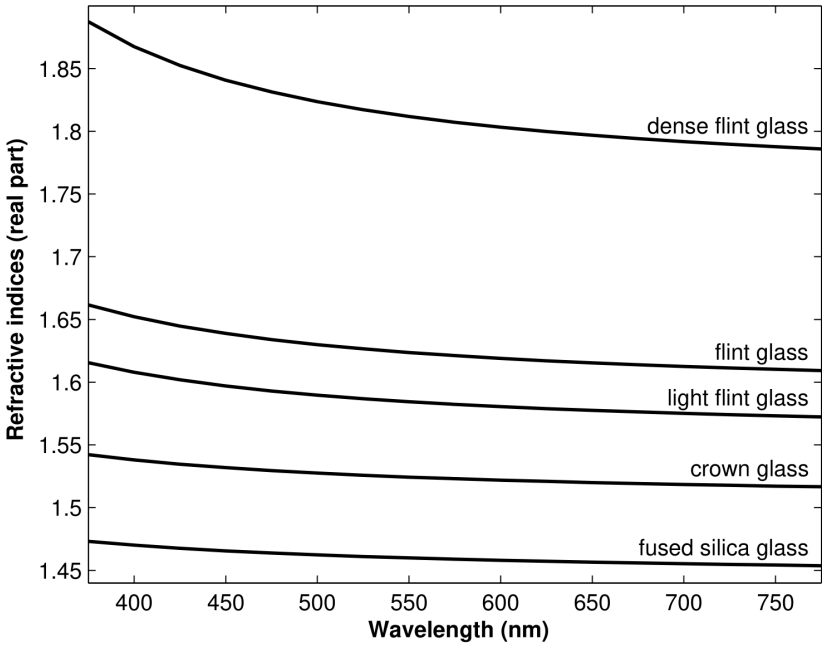


Figure 33: Wavelength something something

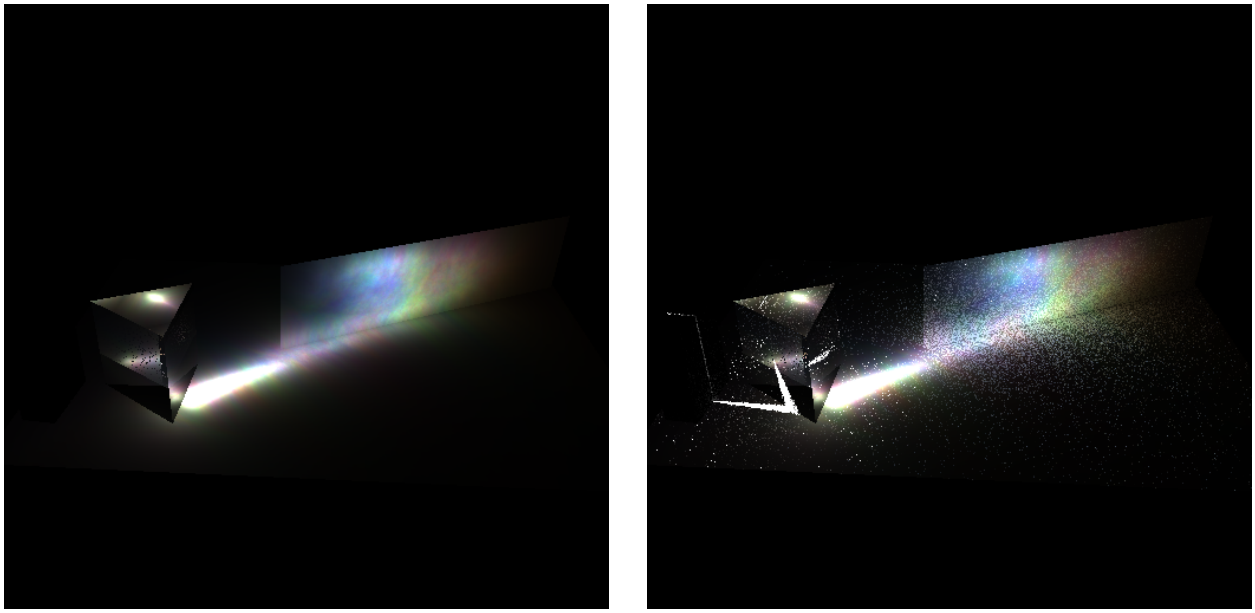


Figure 34: Crown glass. To the left, caustic photons alone. To the right, caustic and global photons

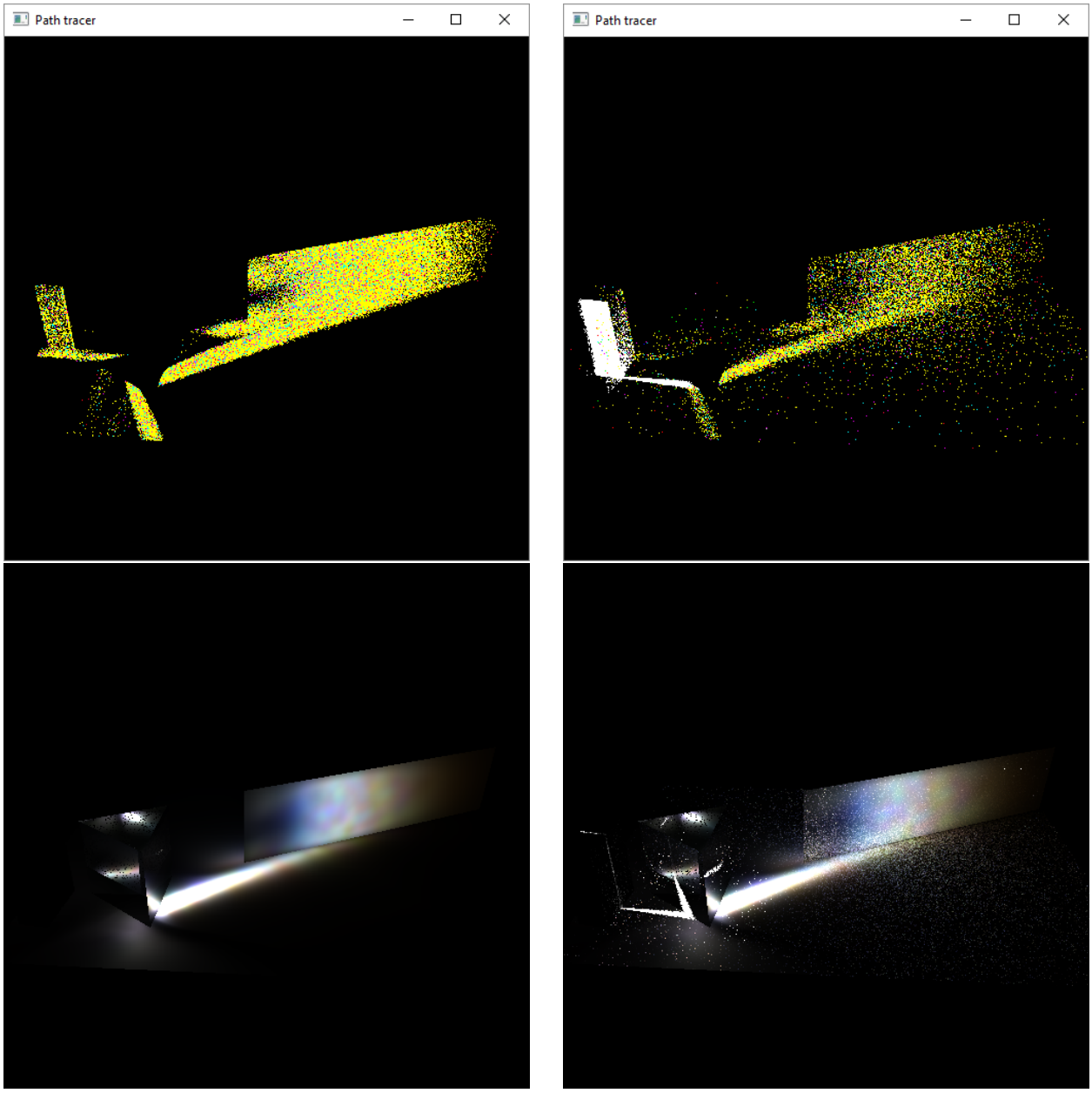


Figure 35: Crown glass. Upper left, photon map of the caustic photons. Upper right, caustic and global photon map. Lower left, caustic photons with improved density estimation. Lower right, caustic and global photons with improved density estimation.

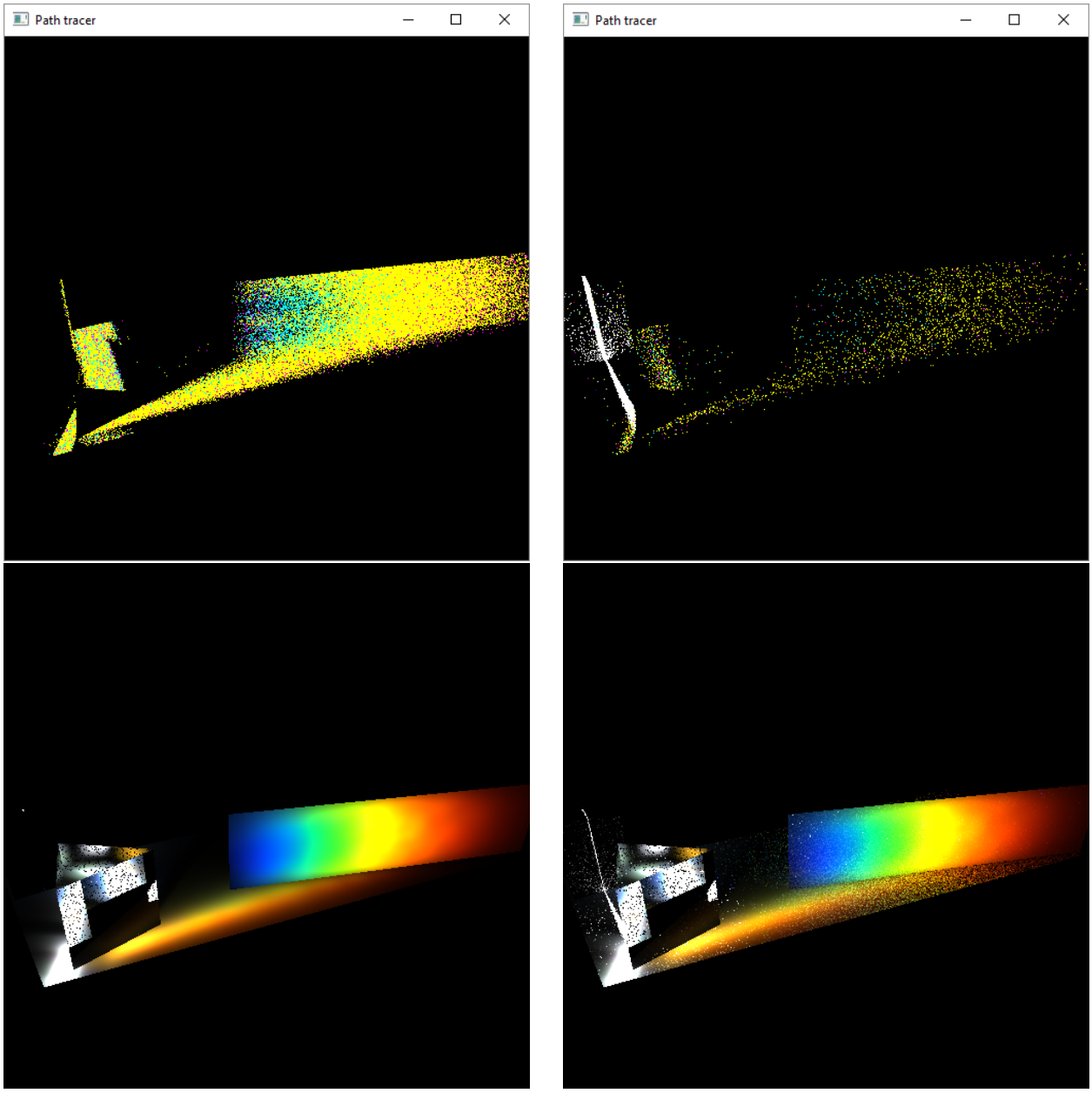


Figure 36: Dense flint glass. Upper left, photon map of the caustic photons. Upper right, caustic and global photon map. Lower left, caustic photons with improved density estimation. Lower right, caustic and global photons with improved density estimation.

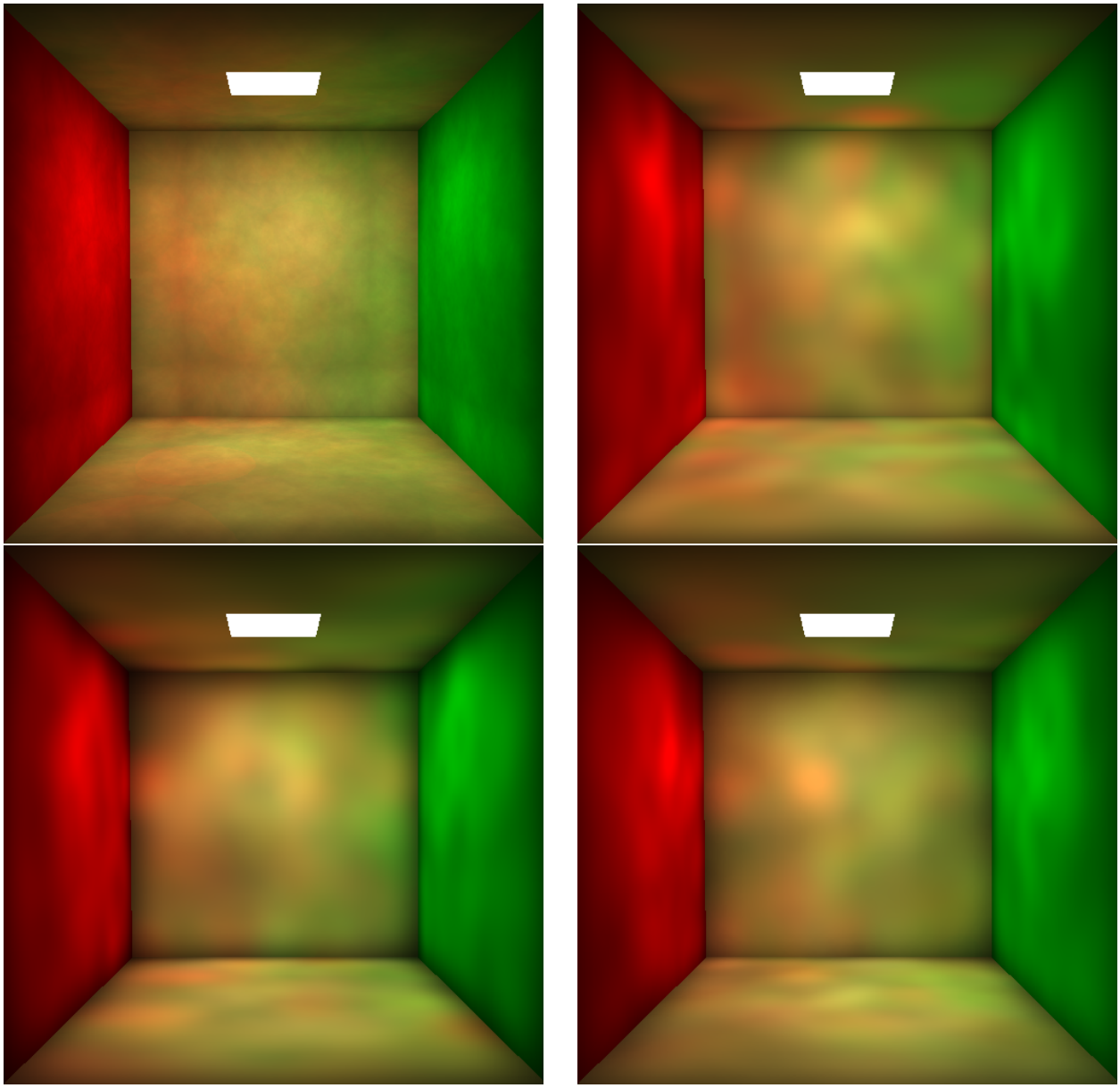


Figure 37: Cornell box rendered using improvements to density estimation. Top left: No improvement. Top right: Simpson kernel. Bottom left: Simpson kernal and normal check. Bottom right: Simpson kernel, normal check and area adjustment using center of mass.

Listing 14: Part 1 ParticleTracer.cpp

```
bool ParticleTracer::disperse_particle(Ray& r, const
    ObjMaterial*& m, int& idx, Vec3f& Phi) const
{
    const bool use_spectral = true;
    float FluxPower = Phi.length();

    // If the ray is not monochromatic (idx < 0), sample a
       monochromatic ray.
    if (idx < 0)
    {
        // Retrieve index of refraction (rgb or spectrum,
           see last argument)
        const Color<complex<double>>& ior = scene->
            get_medium(m).get_ior(use_spectral ? spectrum :
             rgb);
        if (ior.size() == 0) {
            return false;
        }

        // Sample a wavelength and set idx and Phi
        if (use_spectral) {
            double xi1 = mt_random();
            auto i = cdf_bsearch(xi1, spectrum_cdf);
            auto lambda = spectrum_cdf.wavelength + i *
                spectrum_cdf.step_size;
            idx = min(static_cast<int>(ior.size()) - 1, max
                (0, static_cast<int>((lambda - ior.
                wavelength) / ior.step_size)));

            double probability = spectrum_cdf[i];
            if (i > 0) {
                probability -= spectrum_cdf[i - 1];
            }

            Phi = (Phi * normalized_CIE_rgb[i]) /
                probability;
            auto hest = 0;
        }
        else {
            idx = static_cast<int>(mt_random_half_open() *
                3.0);
            float idx_Phi = Phi[idx];
            Phi = Vec3f(0.0f);
            Phi[idx] = idx_Phi * 3.0f;
        }

    }

    //CONTINUES IN NEXT BLOCK
```

Listing 15: Part 2 ParticleTracer.cpp

```
    //CONTINUATION FROM PREVIOUS BLOCK

    // Sample reflection or refraction using Russian
        roulette and set r and m.
    // Return true if something was hit.
    double xi2 = mt_random();
    double R;
    Ray dispersed;
    trace_dispersive(r, dispersed, R, idx, use_spectral ?
        spectrum : rgb);

    if (xi2 < R) {
        Ray reflected;
        if (!trace_reflected(r, reflected)) {
            return false;
        }
        r = reflected;
        m = reflected.get_hit_material();
    }
    else {
        if (!dispersed.has_hit) {
            return false;
        }
        dispersed.inside = dot(dispersed.direction,
            dispersed.hit_normal) > 0.0f;
        r = dispersed;
        m = dispersed.get_hit_material();
    }

    return true;
}
```

Listing 16: PhotonMap.h

```
const CGLA::Vec3f irradiance_estimate(
        const CGLA::Vec3f& pos,              // surface
            position
        const CGLA::Vec3f& normal,           // surface
            normal at pos
        const float max_dist,                // max distance
             to look for photons
        const int nphotons) const            // number of
            photons to use
    {
        NearestPhotons<T> np;
        np.dist2 = (float*)alloca(sizeof(float)*(nphotons +
             1));
        np.index = (const T**)alloca(sizeof(T*)*(nphotons +
             1));

        np.pos = pos;
        np.max = nphotons;
        np.found = 0;
        np.got_heap = 0;
        np.dist2[0] = max_dist*max_dist;

        // locate_photons finds the nearest photons in the
            map given the parameters in np
        locate_photons(&np, 1);

        // sum irradiance from all photons
        CGLA::Vec3f irrad(0.0f);
        CGLA::Vec3f mass_midpoint(0.0f);
        int photon_count = 0;
        for (int i = np.found; i > 0; --i)
        {
            const T* p = np.index[i];
            // the photon_dir call and following if can be
                omitted (for speed)
            // if the scene does not have any thin surfaces
            auto d = pos - p->pos;
            if (dot(photon_dir(p), normal) > 0.0f && abs(
                dot(normalize(d), normal)) < 0.01f)
            {
                ++photon_count;
                mass_midpoint += p->pos;
                float x = np.dist2[i] / np.dist2[0];
                irrad += p->power * 3.0 * ((1.0 - x) * (1.0
                     - x));
            }
        }

        if (photon_count == 0) {
            return CGLA::Vec3f(0.0f);
        }

        mass_midpoint = mass_midpoint / photon_count;

        const float surface_area = M_PI*((np.dist2[0]) -
            sqr_length(mass_midpoint - pos));
        irrad *= 1.0/surface_area;  // estimate of density
        return irrad;
    }
```

8

WORKSHEET 8 DELIVERABLES

We've chosen the Torrance-Sparrow model with the Blinn microfacet distribution. It covers cases a and b from the slides but doesn't handle interreflections. This has the effect that the model in some way becomes too dark when the shininess is set to a low value. By using complex IOR we're able to render a gold material quite nicely. Renderings with local and global illumination can be seen in figure 38. Unfortunately the GI render does not use importance sampling and so had a lot of fireflies around the caustics. We've later corrected this.

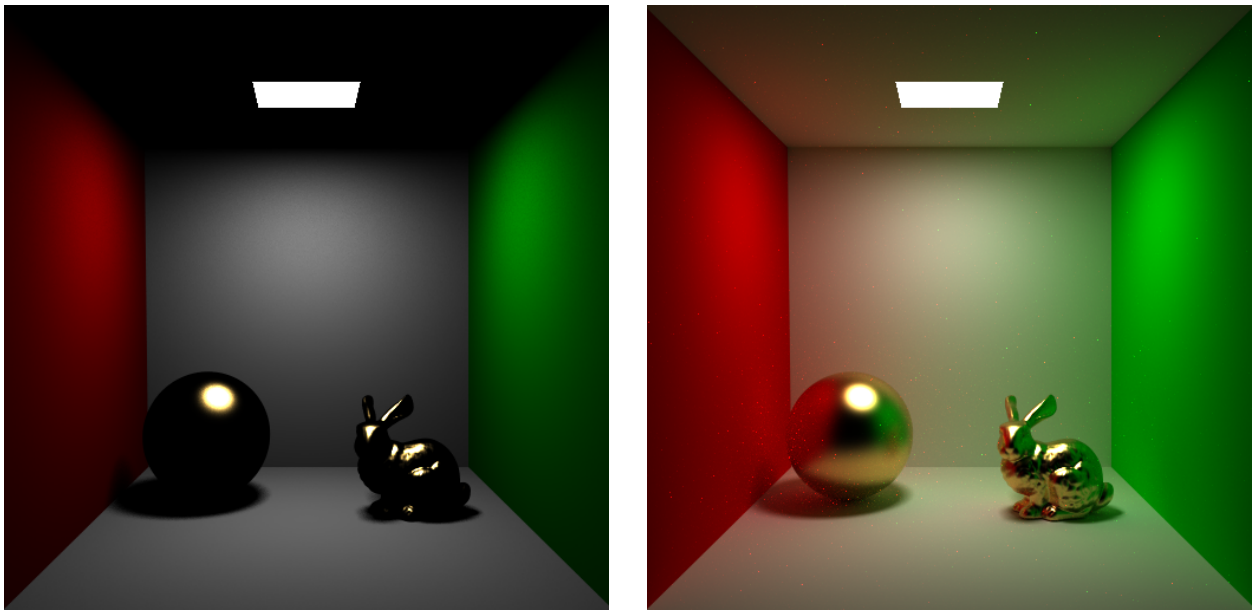


Figure 38: Renders of the microfacet model, both the sphere and the bunny has a shininess of 500, and a gold material. To the left, only local illumination, 261 frames 53 seconds. To the right with global illumination, 141,496 frames 91,391 seconds

Listing 17: First part of `Glossy.cpp`

```
Vec3f Glossy::shade(Ray& r, bool emit) const
{
  using std::max;
  using std::min;
  using std::complex;

  Vec3f rho_d = get_diffuse(r);
  Vec3f rho_s = get_specular(r);
  float s = get_shininess(r);
  int s_int = static_cast<int>(s);
  // Eye vector
  Vec3f dir_out = -r.direction;
  float cos_theta_out = max(dot(r.hit_normal, dir_out), 0.0
      f);
  Vec3f result(0.0f);

  for (auto light : lights) {
    int no_of_samples = light->get_no_of_samples();
    for (int i = 0; i < no_of_samples; i++) {
      Vec3f dir_in, radiance_in;
      if (light->sample(r.hit_pos, dir_in, radiance_in)) {
        Vec3f dir_half = normalize(dir_in + dir_out);

        //auto m = r.get_hit_material();
        //float F = fresnel_R(dot(dir_half, dir_in), r.ior,
            m->ior);
        const auto m1 = tracer->get_hit_medium(r, true);
        const auto m2 = tracer->get_hit_medium(r, false);
        const Color<complex<double>>& ior1 = m1->get_ior(
            rgb);
        const Color<complex<double>>& ior2 = m2->get_ior(
            rgb);

        Vec3f F(0.0f);
        for (size_t i = 0; i < 3; i++) {
          F[i] = fresnel_R(max(0.f, dot(dir_in, dir_half)),
              ior1[i], ior2[i]);
        }

        //CONTINUES IN NEXT BLOCK
```

Listing 18: Second part of `Glossy.cpp`

```
        //CONTINUATION FROM PREVIOUS BLOCK

        float G;
        if (dot(r.hit_normal, dir_out) < dot(r.hit_normal,
            dir_in)) {
          if (2.0f * dot(r.hit_normal, dir_out) * dot(r.
              hit_normal, dir_half) < dot(dir_out, dir_half
              )) {
            G = 2.0f * dot(r.hit_normal, dir_half) / dot(
                dir_out, dir_half);
          } else {
            G = 1.0f / dot(r.hit_normal, dir_out);
          }
        }
        else {
          if (2.0f * dot(r.hit_normal, dir_in) * dot(r.
              hit_normal, dir_half) < dot(dir_out, dir_half
              )) {
            G = (2.0f * dot(r.hit_normal, dir_half) * dot(r
                .hit_normal, dir_in)) / (dot(dir_out,
                dir_half) * dot(r.hit_normal, dir_out));
          } else {
            G = 1.0f / dot(r.hit_normal, dir_out);
          }
        }

        float D1 = int_pow(dot(dir_half, r.hit_normal),
            s_int);
        float D = (s + 2) * 0.5 * M_1_PI * D1;

        float distribution = (s + 2.0f) * 0.5f * M_1_PI *
            int_pow(dot(r.hit_normal, dir_half), s_int);

        Vec3f brdf = (F * G * D) / (4.0f);

        result += brdf * radiance_in;
      }
    }
    // result *= rho_d;
    result /= no_of_samples;
  }

  return result + Emission::shade(r, emit);
}
```

Listing 19: MCGlossy.cpp

```
Vec3f MCGlossy::shade(Ray& r, bool emit) const
{
  return Glossy::shade(r, emit) + estimate(r);
}

Vec3f MCGlossy::estimate(const Ray& r) const
{
  using std::min;
  using std::max;
  using std::complex;

  Ray r_i;
  tracer->trace_Blinn_distribution(r, r_i);
  const Vec3f wi = r_i.direction;
  const Vec3f wo = -r.direction;
  const Vec3f wh = normalize(wo + wi);
  const Vec3f n = r.hit_normal;

  // Second case of G will be chosen if dot(wo, n) = 0,
     thus the whole expression will be 0 in the end.
  if (dot(wo, n) < 10e-6f) {
    return Vec3f(0.f);
  }

  const auto m1 = tracer->get_hit_medium(r, true);
  const auto m2 = tracer->get_hit_medium(r, false);
  const Color<complex<double>>& ior1 = m1->get_ior(rgb);
  const Color<complex<double>>& ior2 = m2->get_ior(rgb);

  Vec3f F(0.0f);
  for (size_t i = 0; i < 3; i++) {
    F[i] = fresnel_R(max(0.f, dot(wi, wh)), ior1[i], ior2[i
        ]);
  }

  float probability = (F[0] + F[1] + F[2]) / 3.f;
  float xi = mt_random();

  if (xi < probability) {
    const float wh_dot_n = dot(wh, n);
    const float G = min({
      wh_dot_n > 10e-6f ? max(0.f, dot(wi, wh)) / wh_dot_n
          : INFINITY,
      2.f * max(0.f, dot(wo, n)),
      2.f * max(0.f, dot(wi, n))
    });

    Vec3f L_i = shade_new_ray(r_i);
    return ((F / dot(wo, n)) * G * L_i) / probability;
  }

  return Vec3f(0.f);
}
```

Listing 20: sampler.h

```
inline CGLA::Vec3f sample_Blinn_distribution(const CGLA::
    Vec3f& normal, const CGLA::Vec3f& dir_o, float
    shininess)
{
  using CGLA::Vec3f;
  using std::max;
  using std::pow;
  using std::sqrt;

  // Get random numbers
  float xi1 = static_cast<float>(mt_random());
  float xi2 = static_cast<float>(mt_random());

  // Calculate sampled half-angle vector as if the z-axis
      were the normal
  float cos_theta_h = pow(xi1, 1.f / (shininess + 2.f));
  float sin_theta_h = sqrt(max(0.f, 1.f - cos_theta_h*
      cos_theta_h));
  float phi = 2.f * M_PI * xi2;
  Vec3f dir_h = spherical_direction(sin_theta_h,
      cos_theta_h, phi);

  // Make sure that the half-angle vector points in the
      right direction
  rotate_to_normal(normal, dir_h);
  float dir_o_dot_dir_h = dot(dir_o, dir_h);
  if (dir_o_dot_dir_h < 0.f) {
    dir_h = -dir_h;
    dir_o_dot_dir_h = -dir_o_dot_dir_h;
  }

  // Return the reflection of "dir" around the half-angle
      vector
  Vec3f dir_i = 2.f * dir_o_dot_dir_h * dir_h - dir_o;
  return dir_i;
}
```

9

WORKSHEET 9 DELIVERABLES

We first implemented direct transmission for rays passing through a volume. The left part of figure 39 shows this.

After switching to the low fat chocolate milk from the wine, and using the same shader, the image to the left in figure 40 appears. A fully black liquid resembling tar or used motor-oil, rather than chocolate milk.

Afterwards we implemented full volume rendering. The result of this can be seen to the right in figures 39 and 40.

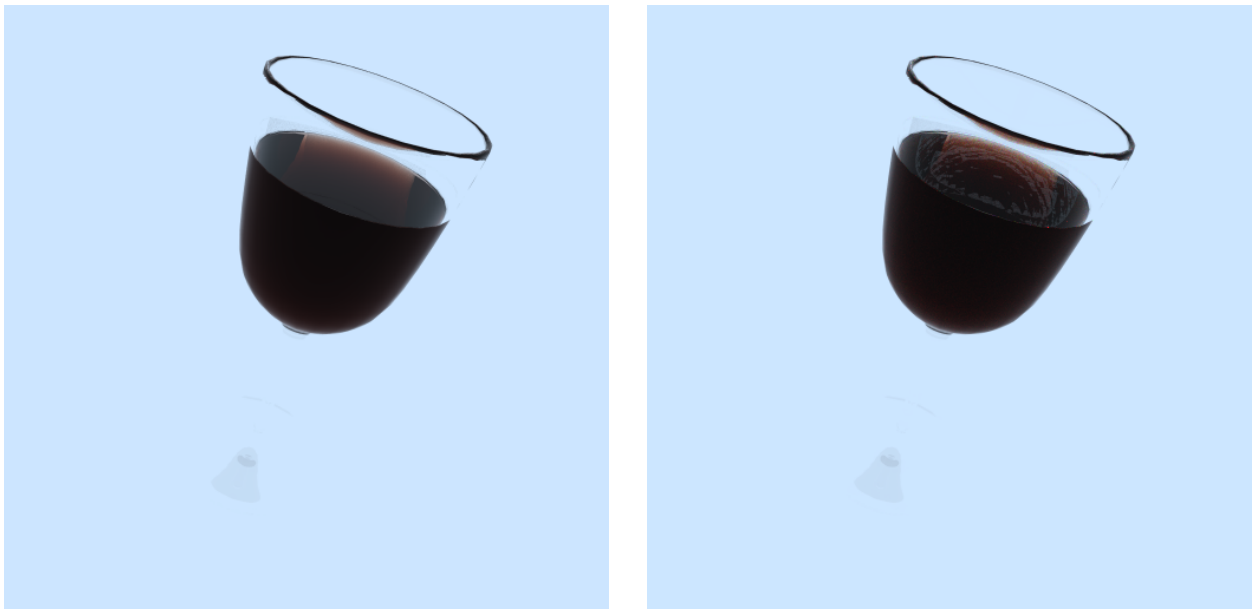


Figure 39: Red wine in glass. To the left, only direct transmission. 681 frames 280 seconds. To the right, full solution. 296 frames 163 seconds.

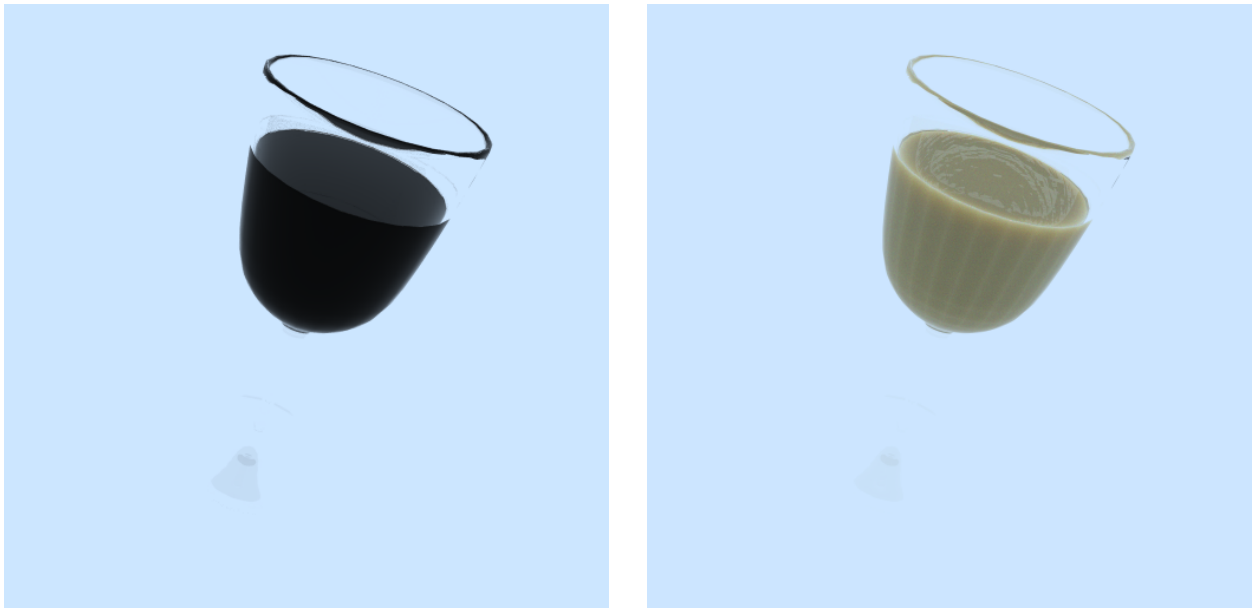


Figure 40: Low fat chocolate milk in glass. To the left, only direct transmission. 106 frames 57 seconds. To the right, full solution. 9,547 frames 12,961 seconds.

Listing 21: Volume.cpp

```
const double Volume::scene_scale = 1.0e-2; // m -> cm

Vec3f Volume::shade(Ray& r, bool emit) const
{
  r.inside = dot(r.direction, r.hit_normal) > 0.0f;
  if (r.trace_depth < splits + 1) // attenuation happens
      after second bounce
    return split_shade(r, emit);

  const Medium* m = tracer->get_medium(r);
  if (m)
  {
    // Compute direct transmission using Russian roulette
        with the transmittance
    // to let absorption stop the recursion.
    return get_transmittance(r, m)*Transparent::shade(r,
        emit);
  }
  return Transparent::shade(r, emit);
}

Vec3f Volume::split_shade(Ray& r, bool emit) const
{
  // Find the direct transmission through the volume by
      using the transmittance
  // to modify the result from the Transparent shader.
  const Medium* m = tracer->get_medium(r);
  return get_transmittance(r, m)*Transparent::shade(r, emit
      );
}

Vec3f Volume::get_transmittance(Ray& r, const Medium* m)
    const
{
  if (m)
  {
    // Compute and return the transmittance using the rgb
        extinction coefficient of the medium.
    // The Medium class holds scattering properties in SI
        units. As most scenes are modeled in
    // centimeters rather than meters, transform the
        extinction coefficient: 1/m -> 1/cm.

    // sigma_t
    auto extinction_color = m->get_extinction(rgb);
    auto cmdist = -scene_scale * r.dist;

    return Vec3f(exp(extinction_color[0] * cmdist), exp(
        extinction_color[1] * cmdist), exp(extinction_color
        [2] * cmdist));
  }
  return Vec3f(1.0f);
}
```

Listing 22: `MCVolume.cpp`

```
Vec3f MCVolume::shade(Ray& r, bool emit) const
{
  r.inside = dot(r.direction, r.hit_normal) > 0.0f;
  const Medium* m = tracer->get_medium(r);
  if (m && m->turbid)
  {
    // Use looping instead of recursion to avoid stack
       space limit
    Vec3f L(0.0f);
    unsigned int i = static_cast<unsigned int>(
       mt_random_half_open()*3.0);
    double ext = m->get_extinction(rgb)[i] * scene_scale;
    double idist = -log(mt_random_open()) / ext;
    if (idist < r.dist)
    {
      double alb = std::min(m->get_albedo(rgb)[i],
         0.999999);
      double g = m->get_asymmetry(rgb)[i];
      Ray rr = r;
      do
      {
        // Evaluate the diffusion term using single sample
           Monte Carlo integration.
        // Importance sample the phase function using
           tracer->trace_HG(...).
        // Break out of the loop if the ray reaches the
           surface or gets absorbed.
        auto xi = mt_random();
        if (xi < alb) {
          Ray r2;
          r2.tmax = -log(mt_random_open()) / ext;
          if (!tracer->trace_HG(rr, r2, idist, g)) {
            idist = r2.tmax;
            rr = r2;
          }
          else {
            L[i] += Transparent::shade(r2)[i];
            break;
          }
        } else {
          break;
        }
      } while (true);
    }
    else {
      L[i] += Transparent::shade(r)[i];
    }

    L[i] *= 3.0f;
    return L;
  }
  return Volume::shade(r, emit);
}
```

Listing 23: sampler.h

```
inline CGLA::Vec3f sample_HG(const CGLA::Vec3f& forward,
    double g)
{
  using std::min;
  using std::max;
  using CGLA::sqr;

  // Get random numbers
  auto xi1 = mt_random_open();
  auto xi2 = mt_random();

  // Importance sample theta according to [Hanrahan and
      Krueger 1992, Pharr and Humphreys 2004; 2010]
  float cos_theta;
  if (abs(g) > 10e-6) {
    cos_theta = (1.0f / (2.0f * g)) * (1.0f + g*g - sqr
        ((1.0f - g*g) / (1.0f - g + 2.0f * g * xi1)));
  }
  else {
    cos_theta = 2.0f * xi1 - 1.0f;
  }
  auto sin_theta = sqrt(max(0.0, 1.0 - cos_theta*cos_theta)
      );

  // Get random phi
  auto phi = 2 * M_PI * xi2;
  auto cos_phi = cos(phi);
  auto sin_phi = sin(phi);

  // Calculate new direction as if the z-axis were the
      forward direction
  CGLA::Vec3f direction(
    cos_phi * sin_theta,
    sin_phi * sin_theta,
    cos_theta
  );

  // Rotate from z-axis to forward direction
  rotate_to_normal(forward, direction);
  return direction;
}
```

Listing 24: PathTracer.cpp

```
bool PathTracer::trace_HG(const Ray& in, Ray& out, double
    distance, double g) const
{
  // Determine the origin and direction of the outgoing ray
  out.origin = in.origin + distance * in.direction;
  out.direction = sample_HG(in.direction, g);
  out.trace_depth = in.trace_depth + 1;
  out.did_hit_diffuse = in.did_hit_diffuse;
  return trace(out);
}
```

10

WORKSHEET 10 DELIVERABLES

Not implemented

11

WORKSHEET 11 DELIVERABLES

We've implemented subsurface rendering, first implementing single scattering contribution (see figure 41) and then the remaining parts (see figure 43).

For comparison the same scene has been rendered using the volume rendering from week 9, see figure 42, as that in some way is the goal result. It has rendered for roughly same amount of time as the final subsurface render.

Figure 44 shows the diffusion part only. It can be seen that the subsurface render is darker than the volume render method, but also has far less variance.

Figure 41: Single shade rendering of the marble block. 128,340 frames 90,646 seconds

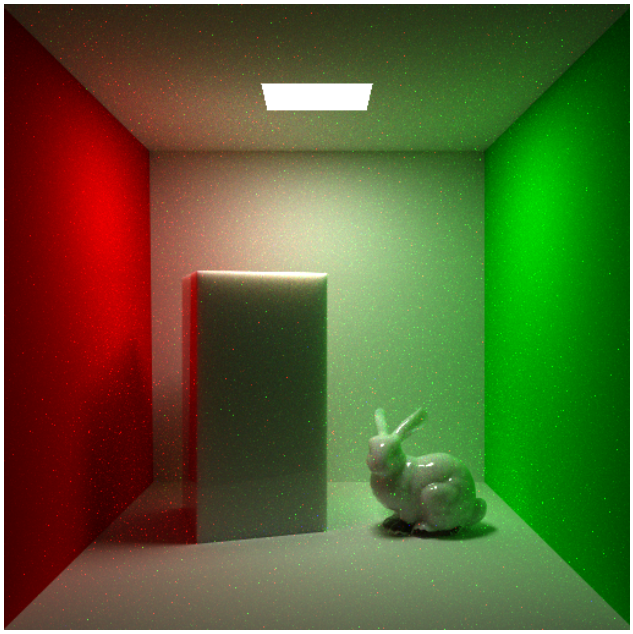


Figure 42: Marbleblock in the cornell box with volume rendering. 37,402 frames 77,172 seconds

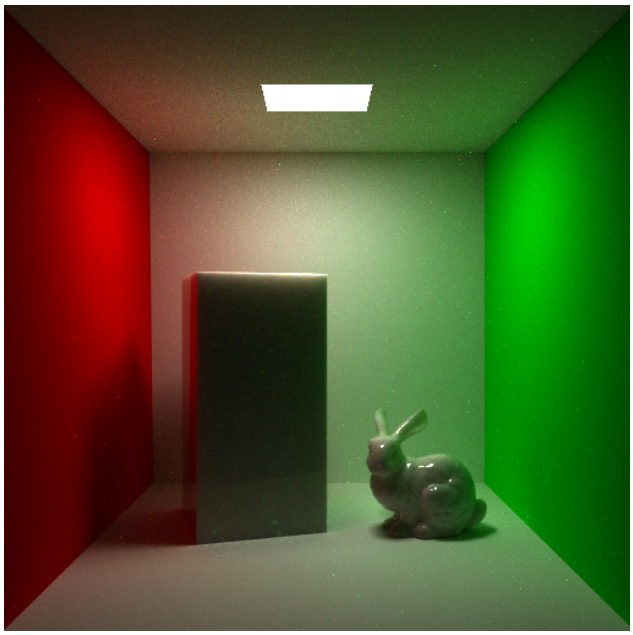


Figure 43: 11,375 frames 77,869 seconds

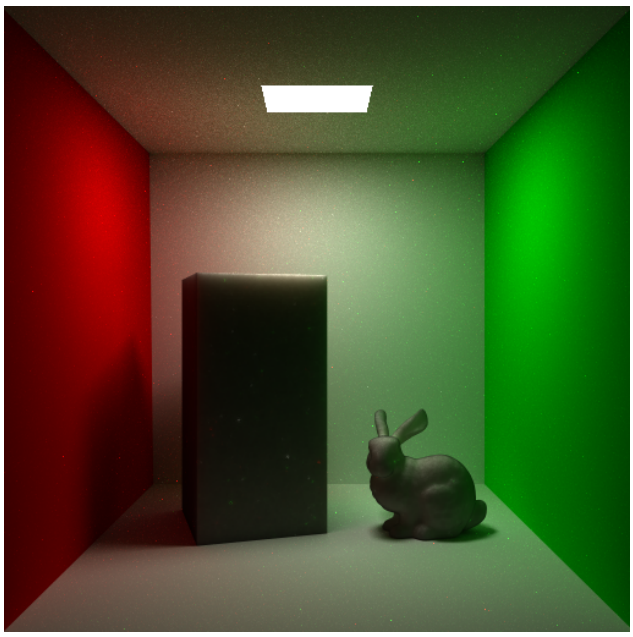


Figure 44: 30,281 frames 66,338 seconds

Listing 25: `single_shade` in `MCSubsurf.cpp`

```
Vec3f MCSubsurf::single_shade(Ray& r, bool emit) const
{
  r.inside = dot(r.direction, r.hit_normal) > 0.0f;
  const Medium* m = MCVolume::tracer->get_medium(r);
  if (m && m->turbid) {
    // Implement Monte Carlo evaluation of single
       scattering.
    Vec3f L(0.0f);
    unsigned int i = static_cast<unsigned int>(
       mt_random_half_open()*3.0);
    double ext = m->get_extinction(rgb)[i] * scene_scale;
    double idist = -log(mt_random_open()) / ext;
    if (idist < r.dist) {
      double alb = std::min(m->get_albedo(rgb)[i],
         0.999999);
      double g = m->get_asymmetry(rgb)[i];
      Ray rr = r;
      auto xi = mt_random();
      if (xi < alb) {
        Ray r2;
        r2.tmax = -log(mt_random_open()) / ext;
        if (!MCVolume::tracer->trace_HG(rr, r2, idist, g))
           {
          idist = r2.tmax;
          rr = r2;
        } else {
          L[i] += Transparent::shade(r2)[i];
        }
      }
    } else {
      L[i] += Transparent::shade(r)[i];
    }

    L[i] *= 3.0f;
    return L;
  }
  return Volume::shade(r, emit);
}
```

Listing 26: First part of `init_sample_surface` in `MCSubsurf.cpp`

```
void MCSubsurf::init_sample_surface(const TriMesh* mesh)
{
  // Init photon map for finding the Poisson distribution
  PhotonMap<> sample_map(samples);
  NearestPhotons<Photon> np;
  np.dist2 = (float*)alloca(sizeof(float) * 3);
  np.index = (const Photon**)alloca(sizeof(Photon*) * 3);
  np.max = 2;
  np.dist2[0] = mesh->surface_area / static_cast<float>(
      samples);

  // Init vector of position samples
  std::vector<PositionSample>& psamples = pos_samples[mesh
      ];
  if (psamples.size() != samples)
    psamples.resize(samples);

  // Get geometry info
  const IndexedFaceSet& geometry = mesh->geometry;
  const IndexedFaceSet& normals = mesh->normals;
  const float no_of_faces = geometry.no_faces();

  for (unsigned int i = 0; i < samples; ++i) {
    PositionSample& sample = psamples[i];
    sample.initialized = false;

    // Sample a triangle face
    sample.hit_face_id = cdf_bsearch(static_cast<float>(
        mt_random()), mesh->face_area_cdf);
    const Vec3i& face = geometry.face(sample.hit_face_id);
    const Interface* iface = MCVolume::tracer->
        get_interface(mesh->materials[mesh->mat_idx[sample.
        hit_face_id]].name);
    const Medium* med_in = iface->med_in->turbid ? iface->
        med_in : iface->med_out;
    const Medium* med_out = iface->med_in->turbid ? iface->
        med_out : iface->med_in;

    // Get triangle vertices
    Vec3f v0 = geometry.vertex(face[0]);
    Vec3f v1 = geometry.vertex(face[1]);
    Vec3f v2 = geometry.vertex(face[2]);

    // Get triangle normals
    Vec3f n0 = normals.vertex(face[0]);
    Vec3f n1 = normals.vertex(face[1]);
    Vec3f n2 = normals.vertex(face[2]);

    // Compute hit position (set sample.pos)
    Vec3f uvw;
    unsigned int counter = 0;
    do {
      // Sample point in triangle
      const float xi1 = mt_random();
      const float xi2 = mt_random();

      uvw[0] = max(0.f, 1.f - sqrt(xi1));
      uvw[1] = max(0.f, 1.f - xi2) * sqrt(xi1);
      uvw[2] = xi2 * sqrt(xi1);

      sample.pos = uvw[0] * v0 + uvw[1] * v1 + uvw[2] * v2;

    //CONTINUES IN NEXT BLOCK
```

Listing 27: Second part of `init_sample_surface` in `MCSubsurf.cpp`

```
//CONTINUATION FROM PREVIOUS BLOCK

  // Check vicinity to get Poisson disk distribution
  if (sample_map.get_photon_count()) {
    np.found = 0;
    np.got_heap = 0;
    np.pos = sample.pos;
    sample_map.locate_photons(&np, 1);
  } else {
    break;
  }
} while (np.found && ++counter < 4);
sample_map.store(Vec3f(0.0f), sample.pos, Vec3f(0.0f));
sample_map.balance();
sample.u = uvw[0];
sample.v = uvw[1];

// Compute hit normal (set sample.normal)
sample.normal = uvw[0] * n0 + uvw[1] * n1 + uvw[2] * n2
    ;

// Trace a sample ray from the sampled position to
    sample the incident radiance
Vec3f wi = sample_cosine_weighted(sample.normal);
Ray r(sample.pos, wi);
MCVolume::tracer->trace(r);

// Find the cosine term where light is incident on the
    surface (cos_theta_i)
float cos_theta_i = dot(wi, sample.normal);

//CONTINUES IN NEXT BLOCK
```

Listing 28: Third part of `init_sample_surface` in `MCSubsurf.cpp`

```
    //CONTINUATION FROM PREVIOUS BLOCK

    // Find the direction of the transmitted ray (
        cos_theta_t)
    double ior1 = (med_out ? med_out->get_ior(mono)[0].real
        () : 1.0);
    double ior2 = med_in->get_ior(mono)[0].real();

    double cos_theta_i_sqr = cos_theta_i*cos_theta_i;
    double sin_theta_t_sqr = ((ior1*ior1) / (ior2*ior2)) *
        (1.0 - cos_theta_i_sqr);

    double cos_theta_t = sqrt(std::max(0.0, 1.0 -
        sin_theta_t_sqr));

    // Compute Frensel transmittance and handle total
        internal reflection
    double R = fresnel_R(cos_theta_i, ior1, ior2);
    double T = 1.0 - R;

    const Color< complex<double> >& ior_i = med_in->get_ior
        (rgb);
    Color< complex<double> > ior_o;
    if (med_out)
      ior_o = med_out->get_ior(rgb);
    else
      ior_o.resize(3, complex<double>(1.0, 0.0));

    Vec3f Li = shade_incident(r, sample.normal);

    // Calculating: (At / A) * (1 / At) = (At / A) / At = 1
         / A
    // float probabilities = (1.f / mesh->surface_area) ;

    // Compute radiance entering the medium at the sampled
        position
    sample.L = T * Li * mesh->surface_area;

    if (dot(sample.L, sample.L) > 0.0f) {
      sample.initialized = true;
    }
  }
  poisson_samples = sample_map.get_photon_count();
}
```

Listing 29: `diffusion` in `MCSubsurf.cpp`

```
Vec3f MCSubsurf::diffusion(double dist, const
    DiffusionProperties& dp, const PositionSample& sample)
    const
{
  // Compute distances to dipole sources
  float zr_sqr = dp.real_source[2];
  float zv_sqr = dp.virtual_source[2];

  float zr = sqrt(zr_sqr);
  float zv = sqrt(zv_sqr);

  float dr = sqrt(dist*dist + zr_sqr);
  float dv = sqrt(dist*dist + zv_sqr);

  // Compute intensities for dipole sources
  Vec3f real_intensity = (zr * (one + dp.transport * dr) *
      Vec3f(exp(-dp.transport[0]*dr), exp(-dp.transport[1]
      * dr), exp(-dp.transport[2] * dr))) / (dr*dr*dr);
  Vec3f virtual_intensity = (zv * (one + dp.transport * dv)
       * Vec3f(exp(-dp.transport[0] * dv), exp(-dp.
      transport[1] * dv), exp(-dp.transport[2] * dv))) / (
      dv*dv*dv);

  // Compute diffusion part of the BSSRDF and evaluate the
      rendering equation
  // (remember to divide by the probabilities of the
      sampled ray directions)

  float probability = dot(sample.dir, sample.normal) *
      M_1_PI;
  Vec3f Rd = 0.25f * M_1_PI * dp.reduced_alb * (
      real_intensity + virtual_intensity);

  return dp.fresnel_T_o * Rd * sample.L;
}
```