Homework 2

Computer Graphics

Francesco Palandra 1849712

November 5, 2021

Contents

1 Refractive
(easy)

For the refractive we used the Snell's law [Wikipedia] to handle the behavior of the light with a refractive object. The direction of the light entering the object is bent towards the normal of the surface proportionally to the ratio of the two η (refraction coefficients). We need to distinguish two case, when the ray is entering into the volume and when is leaving it. To establish between this two case we can simply check the dot product of the direction of the ray and the normal, if it's incoming, the ratio $\frac{\eta_1}{\eta_2}$ where η_1 is the refraction coefficient of the air, so it is equal to 1, and η_2 is the refraction coefficient of the object (ior). If it's outgoing we set the ratio will be just the ior of the element and we also turn the normal of the object to calculate thought the refract function the outgoing direction of the ray.

Algorithm 1 Refractive's algorithm

Input: *ior*
Output: *radiance*
1: **if** $(rand < fresnel(0.4,\ normal,\ ray.d))$ **then**
2: $\quad incoming \leftarrow reflect\ (outgoing,\ normal)$
3: $\quad radiance\ +=\ color\ \cdot\ raytrace()$
4: **else**
5: $\quad$ **if** $(dot(ray.d,\ normal)\ >\ 0)$ **then**
6: $\quad\quad normal \leftarrow -normal$
7: $\quad$ **else**
8: $\quad\quad ior\ \leftarrow\ 1/ior$
9: $\quad$ **end if**
10: $\quad incoming \leftarrow refract\ (outgoing,\ normal,\ ior)$
11: $\quad radiance\ +=\ color\ \cdot\ raytrace()$
12: **end if**
13: **return** *radiance*

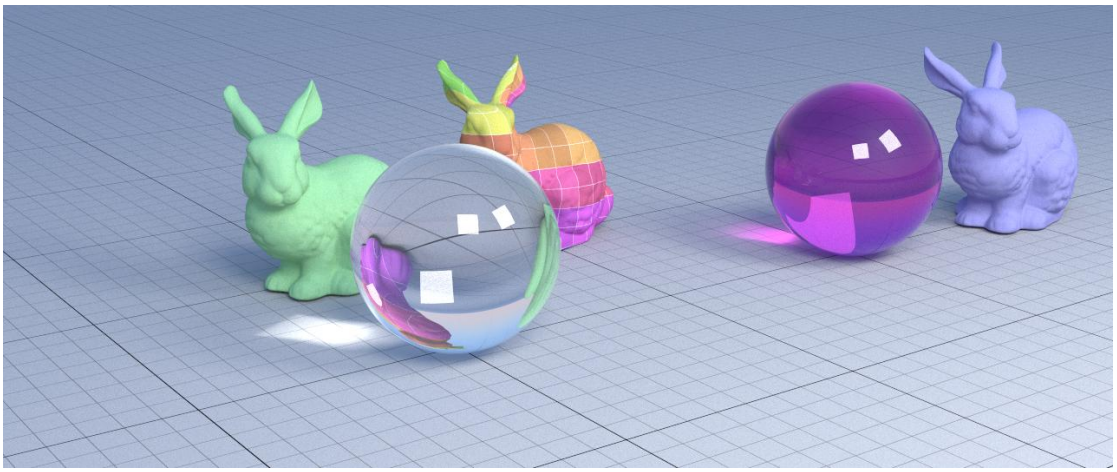

Figure 1: Refractive

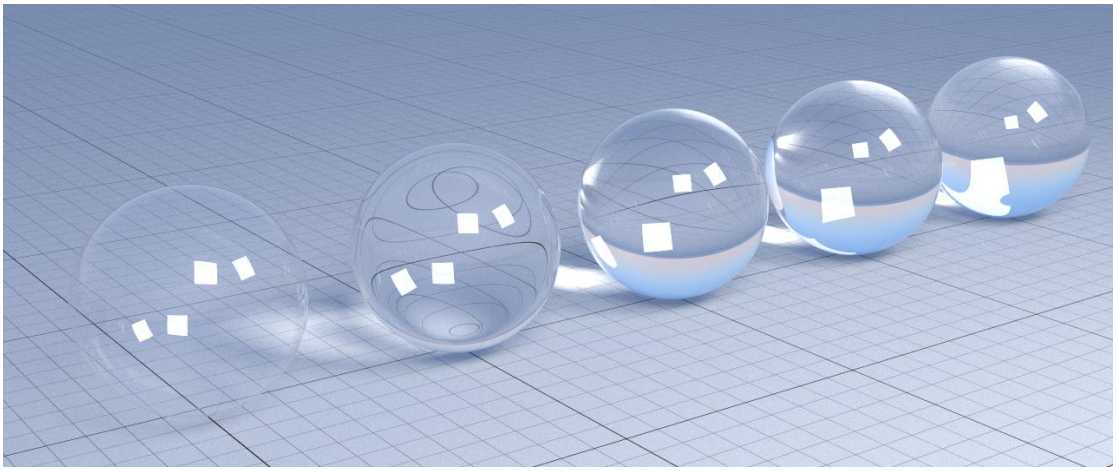

Figure 2: Refractive Sphere with different iors. (1, 1.1, 1.3, 1.4, 1.5)

2 Cell Shading
(medium)

In this shader a toon effect is created handling the colors and the light. Firstly we take the color and through a smooth step function we darken the areas where the light doesn't illuminate. We then create a reflection according to the light position and we create a rim around the object. At the end we calcuate the shadows checking if the ray intersect with the light.

Algorithm 2 Cell Shading algorithm

Input: *scene*
Output: *scene*
1: $lightintensity \leftarrow smoothstep(0, 0.01, dot(light, normal)$
2: $specular \leftarrow reflection(ray, light)$
3: $rim \leftarrow createRim()$
4: **if** $(intersectolight.hit)$ **then**
5: $\quad color* = 0.1$
6: **end if**
7: **return** $color \cdot (specular + lightintensity + ambientcolor + rim)$

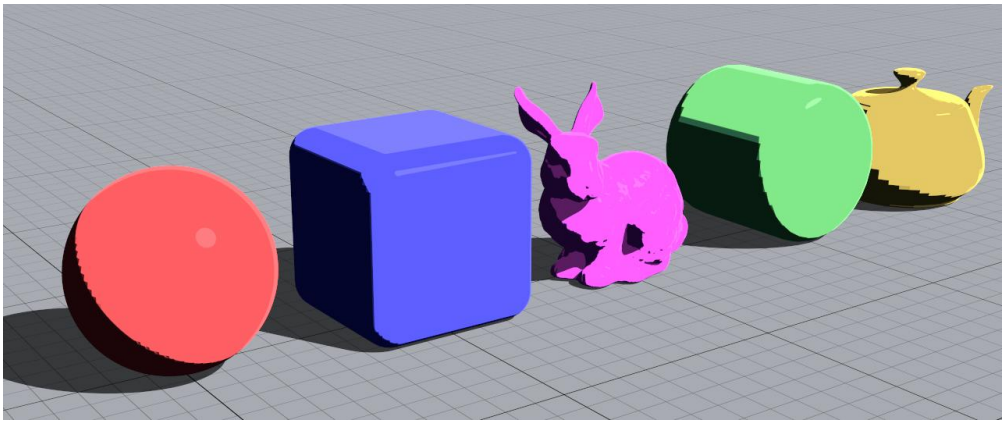

Figure 3: Cell Shading

3 Volume Rendering
(hard)

3.1 Homogeneous Volumes

In homogeneous volumes the density is the same at each point, so we can use the volumetric path tracing [Wikipedia].In volumetric path tracing, a distance between the ray and the surface gets sampled and compared with the distance of the nearest intersection of the ray with the surface. If the sampled distance is smaller, a scatter event occurs. In that case the path gets evaluated and traced from the scatter point in the media. For the scatter it can be use a sample_sphere function that returns a random direction, but the ideal is the Henyey–Greenstein phase function[paper] that ranges from backscattering through isotropic scattering to forward scattering. A problem that occurred on the implementation of this algorithm is handle the absorption, in fact when a sample is inside the volume can both scatter or get absorbed, that depends on the $\sigma_a(absorb\ coefficient)$ and the $\sigma_s(scatter\ coefficient)$. It has been decide for this implementation to ignore the absorption and to scatter on each iteration.

Algorithm 3 Volumetric's algorithm

Input: *ray* σ,
Output: *radiance*

```
outColor ← { 1, 1, 1, 1 }
while (inside the object) do
    if (bounce >= bounces) then
        return { 0, 0, 0, 0 }
    end if
    newisec ← intersect(ray)
    if (!newisec.hit) then
        return enviroment
    end if
    dist_max ← newisec.distance
    dist_rand ← −ln(1−rand)/σ
    if dist_rand < dist_max then
        newpos ← raypoint(ray, dist_rand)
        newdir ← HWDistribuition(rand2)
        newray ← ray{newpos, newdir}
        outColor += color
    else
        radiance += color · raytrace()
    end if
end while
return radiance
```

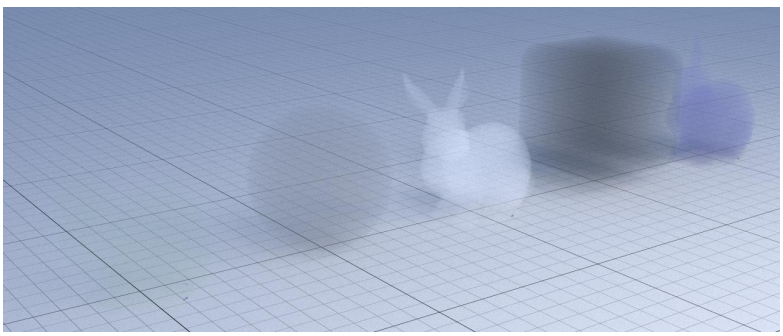

Figure 4: Volumetric path trace with different density

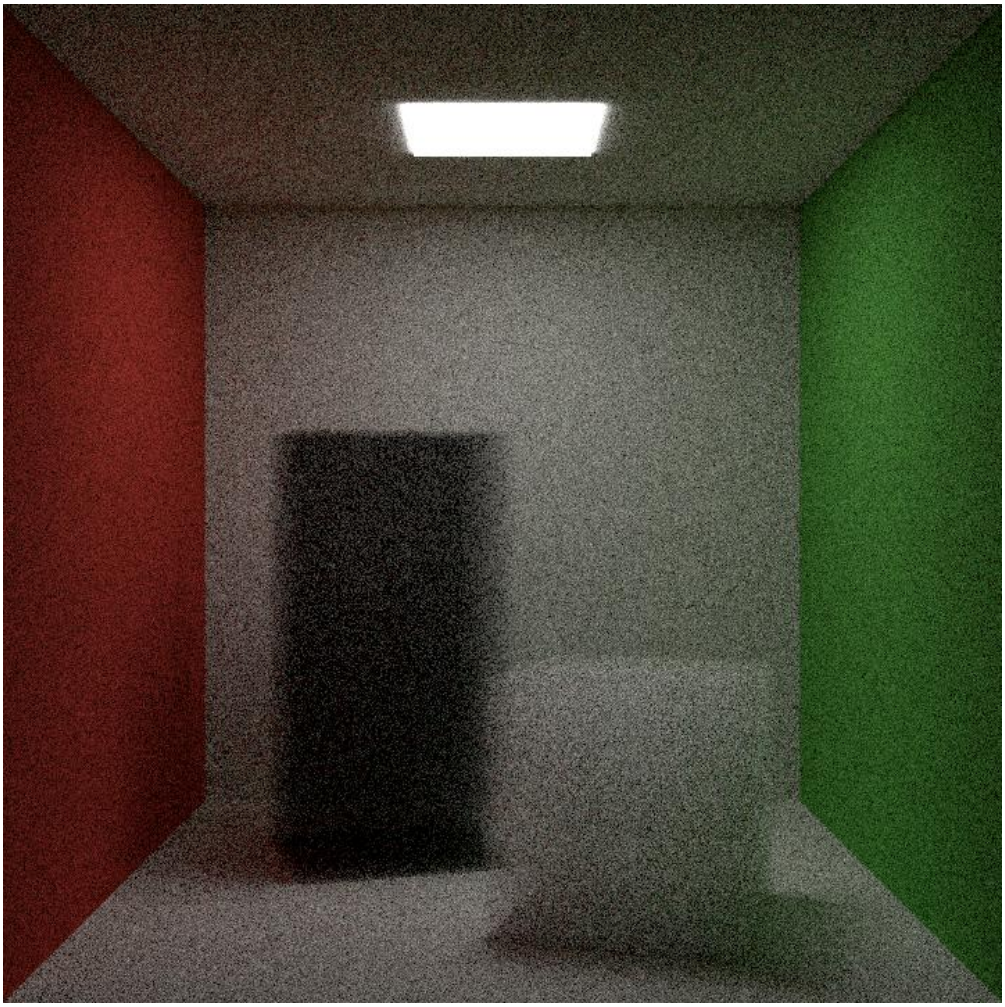

Figure 5: cornell box

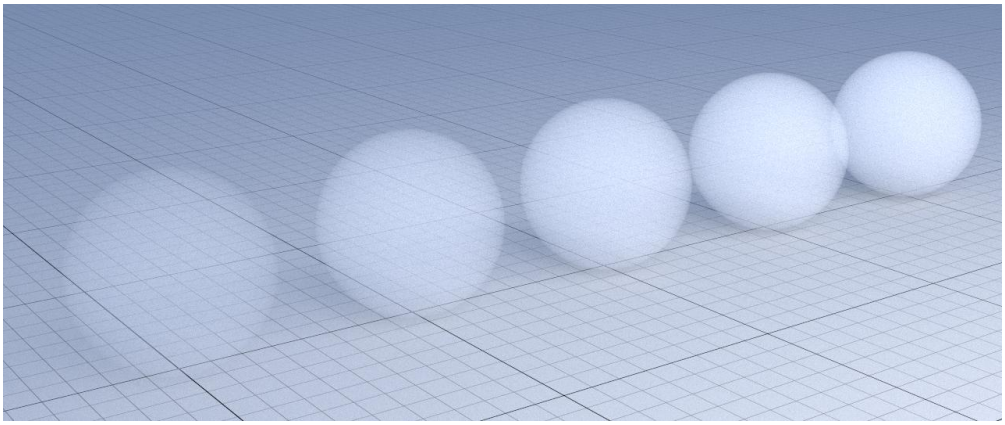

Figure 6: Sphere with different density, from 1.5 to 9.5

3.2 Heterogeneous Volumes

In this homework heterogeneous volumes are not handled, even if two different algorithms will be presented in this section and the results are not from our code, they're just a possible result of the implementation of the two algorithms.

3.2.1 Null Scattering Path Integral

The main issue of the method described in the previous section is that transmittance is not analytically evaluable. As a result, we do not have access to path pdfs and a separate algorithm for heterogeneous media is needed. In this implementation we use a null scattering path integral formulation paper to avoid that problem. Many different implementations are grouped together to implement this algorithm that is taken from here.

Algorithm 4 Medium interaction handling in our null-scattering-aware Uni+NEE MIS volumetric integrator with hero wavelength sampling. f and p are arrays of length 4 containing path throughput and pdf respectively. For other notation, see algorithm 1

Require: *medium*, *ray*, f, p

```
...
x, f_m ← sample_interaction(ray)
f *= f_m, p *= f_m                          ▷ f_m contains only medium transmittance
if x ∈ medium then
    e ← sample_event(x)                     ▷ based on μ_s(x)/μ̄, μ_a(x)/μ̄ and μ_n(x)/μ̄
    if e is absorption then
        f ← 0, p ← 0                        ▷ path termination
    else if e is scattering then
        f *= μ_s(x), p *= μ_s(x)
        ω_e ← sample_emitter_direction(x)   ▷ NEE
        nee_ray ← spawn_ray(x, ω_e)
        f_nee, p_nee, f_uni, p_uni, L_e ← connect_to_light(nee_ray, nee)
        f_p, p_p ← eval_phase(ω_e)
        result += L_e · f·f_nee·f_p / average(p·p_nee·p_p, p·p_uni·p_p)   ▷ balance heuristic
        ω, f_p, p_p ← sample_phase(x)
        update_ray(ray, ω)                  ▷ scatter
        f_nee, p_nee, f_uni, p_uni, L_e ← connect_to_light(ray, uni)
        f_p, p_p ← eval_phase(ω)
        result += L_e · f·f_uni·f_p / average(p·p_nee·p_p, p·p_uni·p_p)   ▷ balance heuristic
    else if e is null then
        f *= μ_n(x), p *= μ_n(x)
        move ray to x
else
    medium escaped
...
```

Algorithm 5 Sampling interactions in the new implementation. Distance is sampled according to hero wavelength.

Require: *ray*

```
t ← -ln(1-ξ) / μ̄^0                          ▷ hero wavelength has index 0
if t is inside medium then
    f ← e^{-t·μ̄}
    return point(ray, t), f
else
    x ← point on the boundary
    d ← x − ray origin
    f ← e^{-d·μ̄}
    return x, f
```

(a) algorithm

(b) Cornell box

Figure 7: Null Scatter Path Integral

3.2.2 Raymarch

In raymarch when a ray hit a volume for each step dL a new ray that connects the new point and the light source is created and we now integrate the density for each step on this new ray as far as we are inside the object. Here a fast implementation that doesn't work, but the results could be like the image above. [Wikipedia]

Algorithm 4 ray march's algorithm

Input: *ray* σ,
Output: *radiance*

```
1:  dL ← 0.001
2:  outColor ← { 0, 0, 0, 0 }
3:  step ← 0.0
4:  while (inside the object) do
5:      position ← ray_point(ray, step)
6:      for eachlight do
7:          lightdir ← normalize(light.o − position)
8:          raytolight ← ray{position, lightdir}
9:          isec ← intersect(raytolight)
10:         lightstep ← 0.0
11:         while (passolight <= isec.dinstance) do
12:             position ← ray_point(raytolight, lightstep)
13:             outColor += color · (1 − totalopacity)
14:             totalopacity += opacity · (1 − totalopacity)
15:             lightstep += dL
16:         end while
17:     end for
18:     step += dL
19: end while
20: return radiance
```

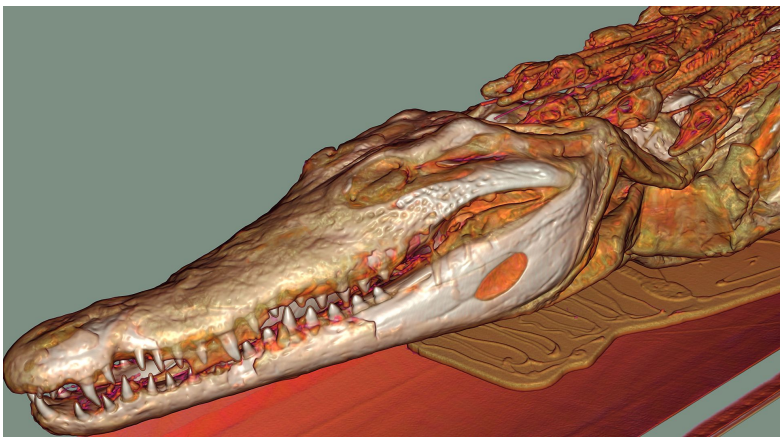

Figure 8: Result of raymarch