DEFERRED ATMOSPHERIC SCATTERING

ÍÑIGO FERNÁNDEZ ARENAS

CS562

INDEX

1. INTRODUCTION 2

1.1 MOTIVATION 2

2. PRINCIPLES OF ATMOSPHERIC SCATTERING 2

2.1 RAYLEIGH SCATTERING 2

2.2 MIE SCATTERING 3

2.3 OPTICAL DEPTH 4

2.4 IN-SCATTERING LIGHT 5

2.5 FINAL LIGHT 5

3. IMPLEMENTATION 6

3.1 REMOVE INNER INTEGRALS 6

3.2 COMPUTE END POINTS 8

3.3 FINAL IMPLEMENTATION 10

3.4 IMPORTANT DETAILS 12

4. RESULTS 13

5. IMPROVEMENTS 14

6. CONCLUSIONS 16

7. BIBLIOGRAPHY 17

1. INTRODUCTION

Atmospheric scattering occurs when a particle in the atmosphere changes the direction of a light ray from its original path. This effect contributes to many everyday phenomena that we experience, and it is the reason the sky is blue at noon and red at sunset. It is also the reason why an object far in the distance appears to have lost a portion of its original color and adopted the color of the sky. In short, atmospheric scattering describes what happens with sunlight when it enters the atmosphere.

1.1 MOTIVATION

While many older sky and haze rendering techniques involve sky cubes and a simple depth-dependent color modifier, it is difficult to create realistic environments and dynamic time of day transitions using these approaches. Instead, it is aimed to provide a way of calculating what is happening with the light emitted by the sun when it enters the atmosphere of a planet.

2. PRINCIPLES OF ATMOSPHERIC SCATTERING

The sun radiates light of all wavelengths in nearly equal intensities and when the sunlight penetrates the atmosphere it gets attenuated due to the various ingredients of the atmosphere which scatter and absorb the sunlight. Scattering of light differs with particle size and varies with wavelength and depending on the size of the particles two types of scattering may be observed inside the atmosphere, which are Rayleigh scattering and Mie scattering.

2.1 RAYLEIGH SCATTERING

Scattering that happens in small particles and molecules in the air of the atmosphere favors short wavelengths, therefore blue colors are much more scattered than other colors. For this reason, Rayleigh scattering is the one responsible for the bluish, reddish color of the sky.

The scattering coefficient for Rayleigh scattering B_R^s can be obtained as:

$$B_R^s(\lambda) = \frac{8\pi^3(n^2-1)^2}{3N\lambda^4}$$

Constants of this equation are n which describes the refractive index of air and N which stands for the molecular density at surface of the planet. Rayleigh scattering is inversely proportional to the 4th power of the wavelength λ, which explains the strong attenuation of short wavelength light. In our implementation we will use a $\boldsymbol{B_R^s}$ **= (0.0058f, 0.0135f, 0.0331f)**.

The extinction coefficient $\boldsymbol{B_R^e}$ determines how much light is scattered or absorbed, but since air molecules doesn't absorb light, only scattered is considered, which gives a $\boldsymbol{B_R^e = B_R^s}$.

Since air molecules doesn't scatter light in every direction with equal intensity a phase function needs to be applied in order to obtain an approximated intensity for the ray scattered:

$$p_R(\emptyset) = \frac{3}{16\pi} * (1 + cos^2(\emptyset))$$

- $\emptyset : angle\ between\ light\ direction\ and\ view\ ray$

2.2 MIE SCATTERING

Scattering that happens in greater particles like aerosols in the atmosphere, it is almost not wavelength dependent, and it is responsible of producing fog, glare around the sun and other effects.

The scattering coefficient for Mie scattering B_M^s can be obtained as:

$$B_M^s = \frac{8\pi^3(n^2-1)^2}{3N}$$

The equation is similar to the Rayleigh equation without taking into account the wavelength since it is not dependent to it. In our implementation we will use $\boldsymbol{B_M^s}$ **= (0.02f, 0.02f, 0.02f).** But it is a value that the user can change in order to obtain different results.

Aerosols also absorb parts of the incident light. The corresponding extinction coefficient of aerosols is the sum of an absorption coefficient B_M^a and the scattering coefficient: $B_M^e = B_M^a + B_M^s$. In our implementation we will approximate $\boldsymbol{B_M^e = \frac{B_M^s}{0.9}}$.

The relative intensity scattered will fall off drastically if the scattering angle differs slightly from the incident direction therefore, we need to apply a phase function to approximate this property:

$$p_M(\emptyset) = \frac{1}{4\pi} * \frac{3 * (1 - g^2)}{2 * (2 + g^2)} * \frac{(1 + cos^2(\emptyset))}{(1 + g^2 - 2g * cos(\emptyset))^{3/2}}$$

- $\emptyset : angle\ between\ light\ direction\ and\ view\ ray$
- $g : constant\ value, 0.76$

2.3 OPTICAL DEPTH

The Optical Depth describes the optical thickness of a medium and is a measure of light transparency over a given path. It is dependent on the atmospheric density, which decreases toward the top boundary exponentially. We have different optical depths for Rayleigh and Mie:

$$\sigma_R(x) = e^{-\frac{h(x)}{H_R}} \qquad \sigma_M(x) = e^{-\frac{h(x)}{H_M}}$$

- $h : distance\ from\ x\ to\ the\ planet\ radius$
- $H_{RM}: Scale\ height\ that\ varies\ the\ density\ ratio\ between\ the\ planet\ radius\ and\ the\ atmosphere\ radius$

The optical depth from a point A to a point B can be calculated by integrating the extinction coefficients and the density ratio over the path:

$$T(A \to B) = \int_A^B \left(B_R^e * e^{-\frac{h(t)}{H_R}} + B_M^e * e^{-\frac{h(t)}{H_M}} \right) * dt$$

The optical depth can be used to obtain the transmittance of the medium and so the extinction factor which can be understood as the fraction or percentage of an incident light that remains after traversing the atmospheric medium over a given path:

$$F_{ex}(A \to B) = e^{-T(A \to B)}$$

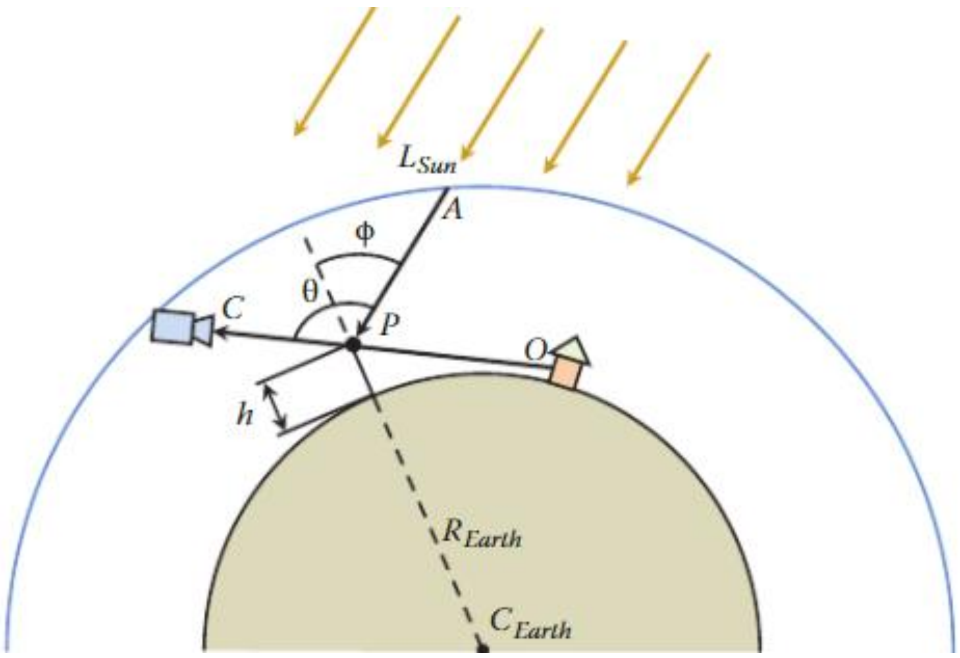


Figure 1. Scattering in the atmosphere towards the camera.

2.4 IN-SCATTERING LIGHT

The in-scattering light represents the amount of light that is redirected from a particle towards the camera. The total in-scattering light over a path can be computed by the following equation:

$$L_{in} = \int_C^O L_{sun} * F_{ex}(A(s) \to P(s)) * F_{ex}(P(s) \to C) * V\big(P(s)\big) *$$

$$* \left(B_R^s * e^{-\frac{h(s)}{H_R}} * p_R(\emptyset) + B_M^s * e^{-\frac{h(s)}{H_M}} * p_M(\emptyset) \right) ds$$

- L_{sun} : *sun light scalated by an sun intensity factor*
- $P(s)$: *Current point in the raymarching*
- $A(s)$: *Intersection point in the atmosphere with the sun direction towards* $P(s)$
- $V\big(P(s)\big)$: *Value that determines if* $P(s)$ *is in shadow* $(0$ *if in shadow*, 1 *if not*$)$

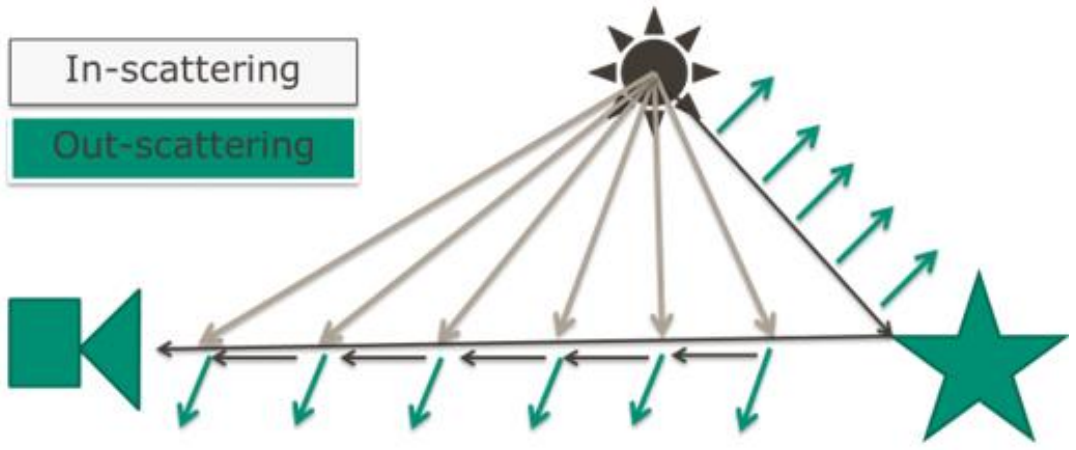


Figure 2. In-scattered and out-scatter over a ray.

2.5 FINAL LIGHT

The final light that reaches the camera will be the sum of the in-scatter light to the reflected light attenuated by the extinction over the path to the camera:

$$L = L_o * F_{ex}(O \to C) + L_{in}$$

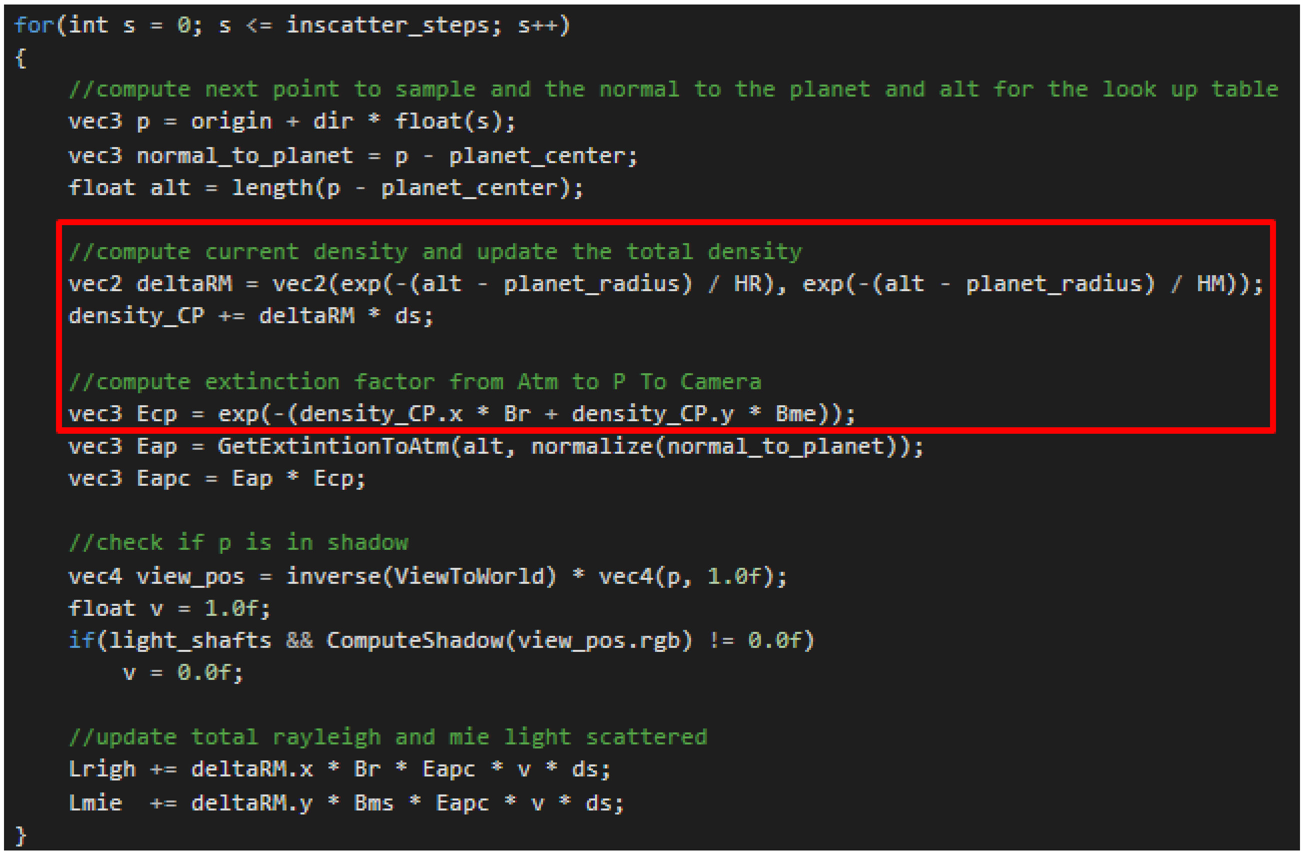
The light reflected by the surface will be computed considering the diffuse and specular value and the sun light attenuated by the path from the top of the atmosphere to the surface point.

3. IMPLEMENTATION

The in-scatter integral cannot be solved numerically in real time, since it has two integrals inside of it. Therefore, we need to somehow get rid of those two integrals that compute the extinction factor over the path and the extinction factor from the sample to the atmosphere. Once we transform the in-scattering integral into a single integral we can solve it by numerical integration which corresponds to a raymarching algorithm. In order to compute the end points of the integral we need to perform ray vs sphere intersection with the planet Since we are using a deferred shading framework, we will do raymarching from every pixel in the screen computing the final light for every pixel.

3.1 REMOVE INNER INTEGRALS

- $\boldsymbol{F_{ex}(P(s) \rightarrow C)}$ **:** the optical depth is computed by maintaining net particle densities from the camera to the current point during the numerical integration process, we can do this because both integrals follow the same direction, so we can compute them at the same time.



```
for(int s = 0; s <= inscatter_steps; s++)
{
    //compute next point to sample and the normal to the planet and alt for the look up table
    vec3 p = origin + dir * float(s);
    vec3 normal_to_planet = p - planet_center;
    float alt = length(p - planet_center);

    //compute current density and update the total density
    vec2 deltaRM = vec2(exp(-(alt - planet_radius) / HR), exp(-(alt - planet_radius) / HM));
    density_CP += deltaRM * ds;

    //compute extinction factor from Atm to P To Camera
    vec3 Ecp = exp(-(density_CP.x * Br + density_CP.y * Bme));
    vec3 Eap = GetExtintionToAtm(alt, normalize(normal_to_planet));
    vec3 Eapc = Eap * Ecp;

    //check if p is in shadow
    vec4 view_pos = inverse(ViewToWorld) * vec4(p, 1.0f);
    float v = 1.0f;
    if(light_shafts && ComputeShadow(view_pos.rgb) != 0.0f)
        v = 0.0f;

    //update total rayleigh and mie light scattered
    Lrigh += deltaRM.x * Br * Eapc * v * ds;
    Lmie  += deltaRM.y * Bms * Eapc * v * ds;
}
```

Figure 3. Raymarching for loop that computes the total in-scatter light.

- $\boldsymbol{F_{ex}(A(s) \rightarrow P(s))}$ **:** Since this integral only depends on the altitude h and an angle φ between the normal to the surface and the sun direction, we can precompute a lookup table storing this information, which in essence is a texture. The first dimension, *x*, takes a sample point at a specific altitude above the planet, with 0.0 being on the ground and 1.0 being at the top of the atmosphere. The second dimension, *y*, represents a vertical angle, with 0.0 being straight up and 1.0 being straight down.

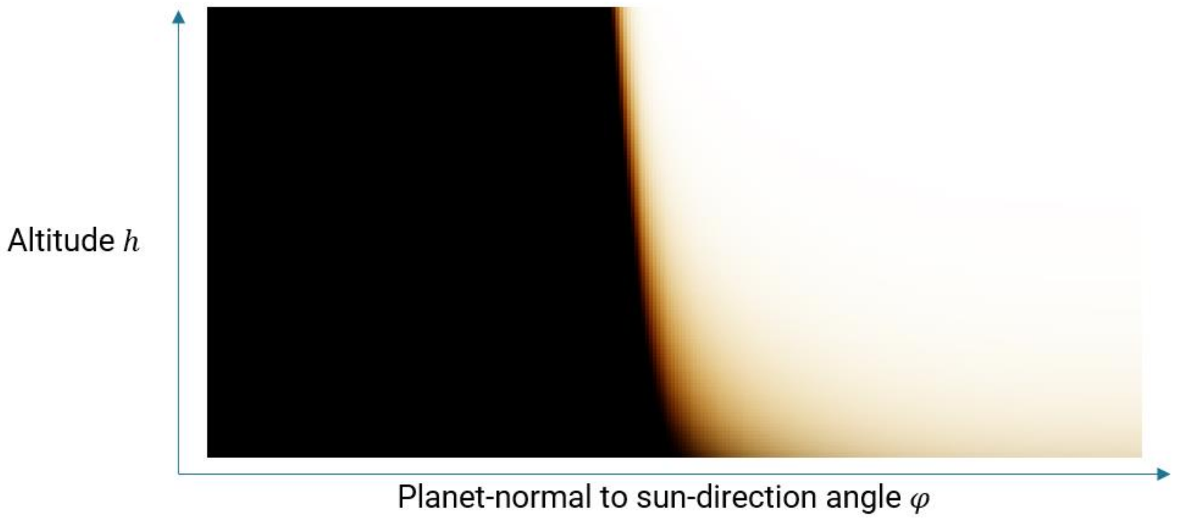
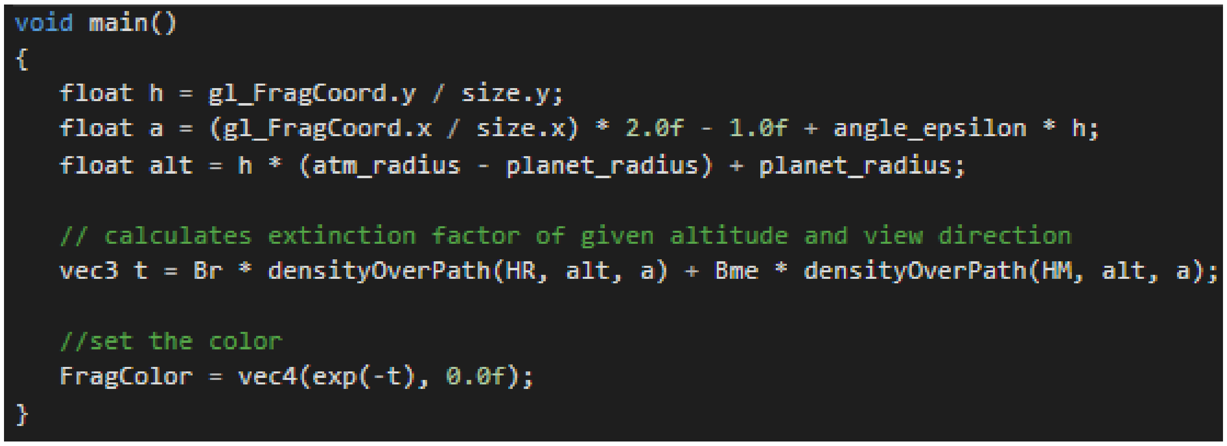


Figure 4. Extinction Texture.

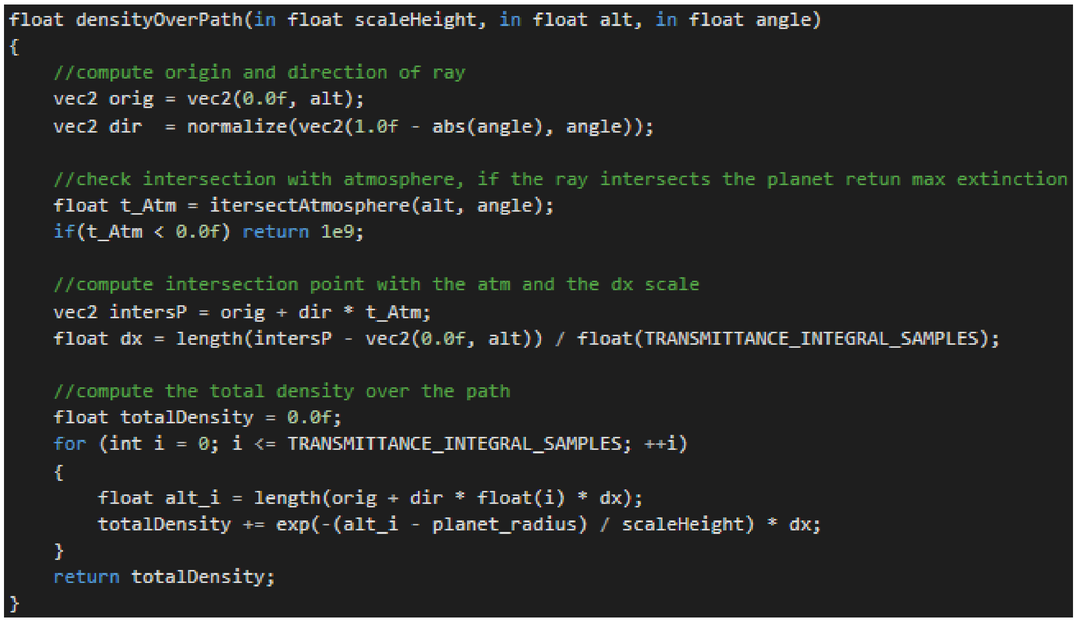
In order to precompute this texture, we will need to do a render pass at the beginning of the application and perform the operations in a shader. First, for each pixel in the texture we need to compute the altitude and the angle, we will add a small epsilon to the angle to avoid artifacts in the simulation:



```
void main()
{
   float h = gl_FragCoord.y / size.y;
   float a = (gl_FragCoord.x / size.x) * 2.0f - 1.0f + angle_epsilon * h;
   float alt = h * (atm_radius - planet_radius) + planet_radius;

   // calculates extinction factor of given altitude and view direction
   vec3 t = Br * densityOverPath(HR, alt, a) + Bme * densityOverPath(HM, alt, a);

   //set the color
   FragColor = vec4(exp(-t), 0.0f);
}
```

Figure 5. Main function to compute the extinction texture.

Then we need to compute the total density over a path for the Rayleigh and Mie extinction factors.



```
float densityOverPath(in float scaleHeight, in float alt, in float angle)
{
    //compute origin and direction of ray
    vec2 orig = vec2(0.0f, alt);
    vec2 dir  = normalize(vec2(1.0f - abs(angle), angle));

    //check intersection with atmosphere, if the ray intersects the planet retun max extinction
    float t_Atm = itersectAtmosphere(alt, angle);
    if(t_Atm < 0.0f) return 1e9;

    //compute intersection point with the atm and the dx scale
    vec2 intersP = orig + dir * t_Atm;
    float dx = length(intersP - vec2(0.0f, alt)) / float(TRANSMITTANCE_INTEGRAL_SAMPLES);

    //compute the total density over the path
    float totalDensity = 0.0f;
    for (int i = 0; i <= TRANSMITTANCE_INTEGRAL_SAMPLES; ++i)
    {
        float alt_i = length(orig + dir * float(i) * dx);
        totalDensity += exp(-(alt_i - planet_radius) / scaleHeight) * dx;
    }
    return totalDensity;
}
```

Figure 6. Function that computes the total density over a path.

3.2 COMPUTE END POINTS

In order to compute the end points of the ray marching integration $O \rightarrow C$ we need to perform ray vs sphere intersection test. We will have two different scenarios:

- **Camera Inside Atm**.
    - O: camera position.
    - C: surface point or top of the atmosphere.
- **Camera Outside Atm**.
    - O: entry point of the view ray with the atmosphere.

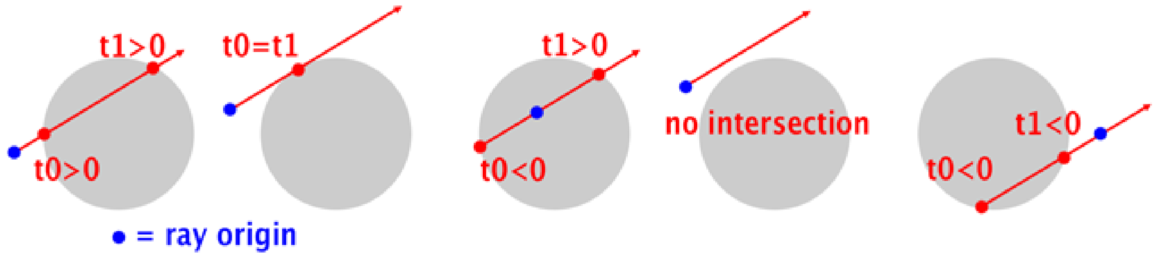
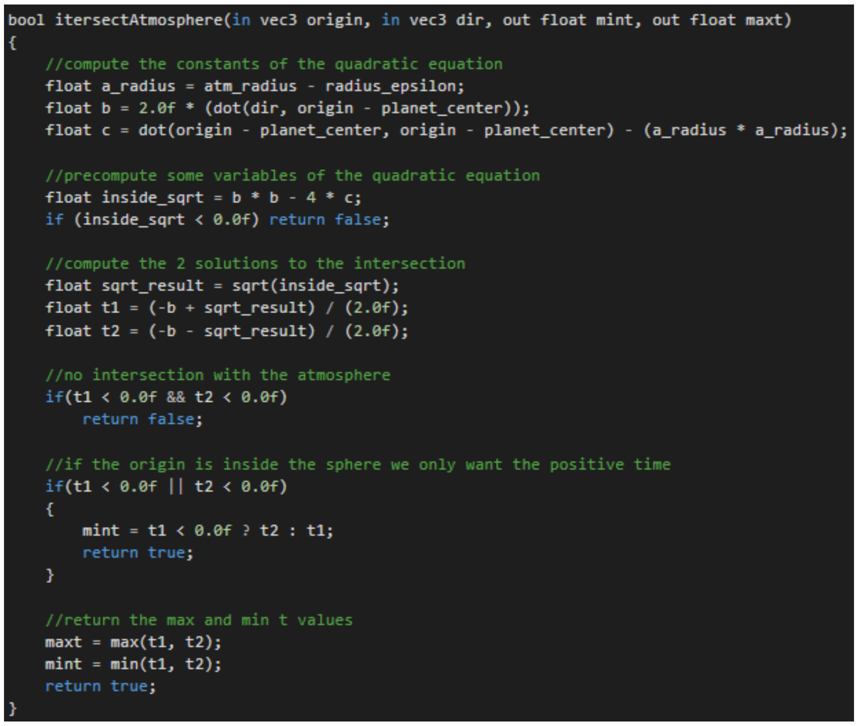


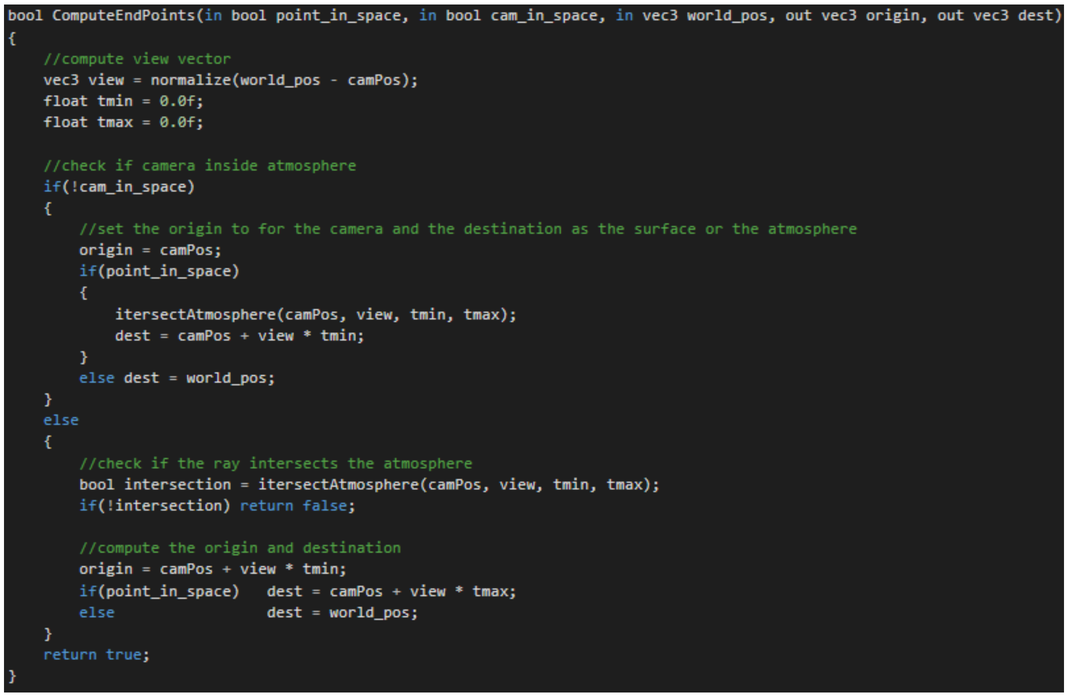
Figure 7. Ray vs sphere intersection test.

- C: surface point of the planet or the exit point of the view ray with the atmosphere.
- $Extra$: the view ray may not intersect the planet therefore no integration will be computed.



```
bool itersectAtmosphere(in vec3 origin, in vec3 dir, out float mint, out float maxt)
{
    //compute the constants of the quadratic equation
    float a_radius = atm_radius - radius_epsilon;
    float b = 2.0f * (dot(dir, origin - planet_center));
    float c = dot(origin - planet_center, origin - planet_center) - (a_radius * a_radius);

    //precompute some variables of the quadratic equation
    float inside_sqrt = b * b - 4 * c;
    if (inside_sqrt < 0.0f) return false;

    //compute the 2 solutions to the intersection
    float sqrt_result = sqrt(inside_sqrt);
    float t1 = (-b + sqrt_result) / (2.0f);
    float t2 = (-b - sqrt_result) / (2.0f);

    //no intersection with the atmosphere
    if(t1 < 0.0f && t2 < 0.0f)
        return false;

    //if the origin is inside the sphere we only want the positive time
    if(t1 < 0.0f || t2 < 0.0f)
    {
        mint = t1 < 0.0f ? t2 : t1;
        return true;
    }

    //return the max and min t values
    maxt = max(t1, t2);
    mint = min(t1, t2);
    return true;
}
```

Figure 7. Function that performs ray vs atmosphere intersection test.

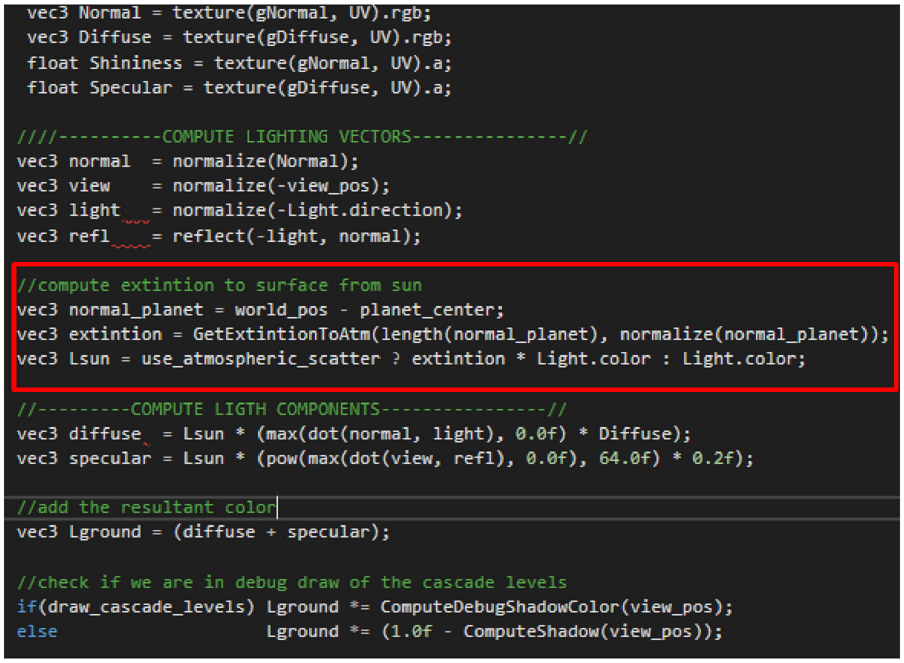


```
bool ComputeEndPoints(in bool point_in_space, in bool cam_in_space, in vec3 world_pos, out vec3 origin, out vec3 dest)
{
    //compute view vector
    vec3 view = normalize(world_pos - camPos);
    float tmin = 0.0f;
    float tmax = 0.0f;

    //check if camera inside atmosphere
    if(!cam_in_space)
    {
        //set the origin to for the camera and the destination as the surface or the atmosphere
        origin = camPos;
        if(point_in_space)
        {
            itersectAtmosphere(camPos, view, tmin, tmax);
            dest = camPos + view * tmin;
        }
        else dest = world_pos;
    }
    else
    {
        //check if the ray intersects the atmosphere
        bool intersection = itersectAtmosphere(camPos, view, tmin, tmax);
        if(!intersection) return false;

        //compute the origin and destination
        origin = camPos + view * tmin;
        if(point_in_space)   dest = camPos + view * tmax;
        else                 dest = world_pos;
    }
    return true;
}
```

Figure 8. Function that computes the end points of the ray marching.

3.3 FINAL IMPLEMENTATION

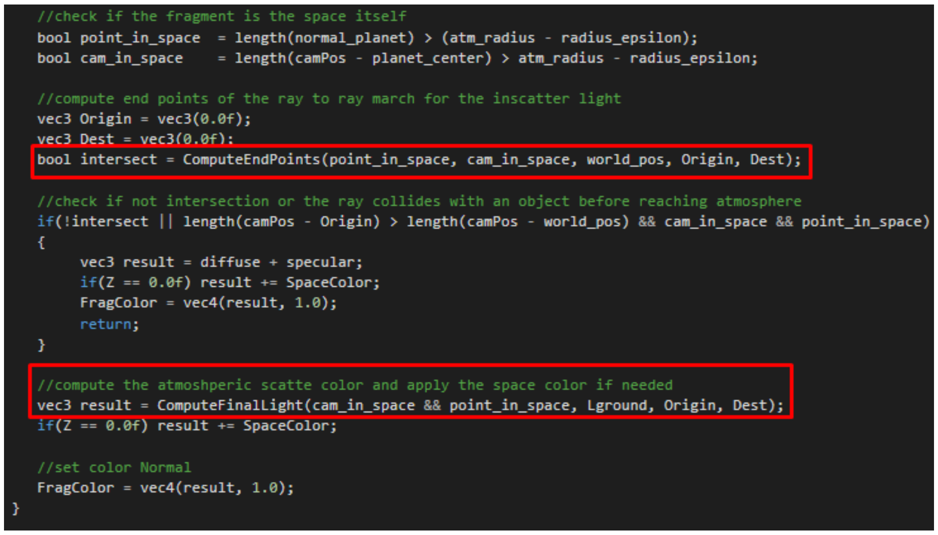
Once we have the precomputations and know how to compute the end points, we need to implement the actual algorithm, for that we will modify the shader used to render a directional light, but once we have the normal color of a normal directional light shading we will attenuate it by an extinction factor from that point to the top of the atmosphere, and pass that resulting color as the L_O for the final atmospheric scattering light integration:



```
 vec3 Normal = texture(gNormal, UV).rgb;
 vec3 Diffuse = texture(gDiffuse, UV).rgb;
 float Shininess = texture(gNormal, UV).a;
 float Specular = texture(gDiffuse, UV).a;

////----------COMPUTE LIGHTING VECTORS---------------//
vec3 normal  = normalize(Normal);
vec3 view    = normalize(-view_pos);
vec3 light   = normalize(-Light.direction);
vec3 refl    = reflect(-light, normal);

//compute extintion to surface from sun
vec3 normal_planet = world_pos - planet_center;
vec3 extintion = GetExtintionToAtm(length(normal_planet), normalize(normal_planet));
vec3 Lsun = use_atmospheric_scatter ? extintion * Light.color : Light.color;

//---------COMPUTE LIGTH COMPONENTS----------------//
vec3 diffuse  = Lsun * (max(dot(normal, light), 0.0f) * Diffuse);
vec3 specular = Lsun * (pow(max(dot(view, refl), 0.0f), 64.0f) * 0.2f);

//add the resultant color
vec3 Lground = (diffuse + specular);

//check if we are in debug draw of the cascade levels
if(draw_cascade_levels) Lground *= ComputeDebugShadowColor(view_pos);
else                    Lground *= (1.0f - ComputeShadow(view_pos));
```

Figure 9. Computing Lo as a normal shading with an extinction factor.

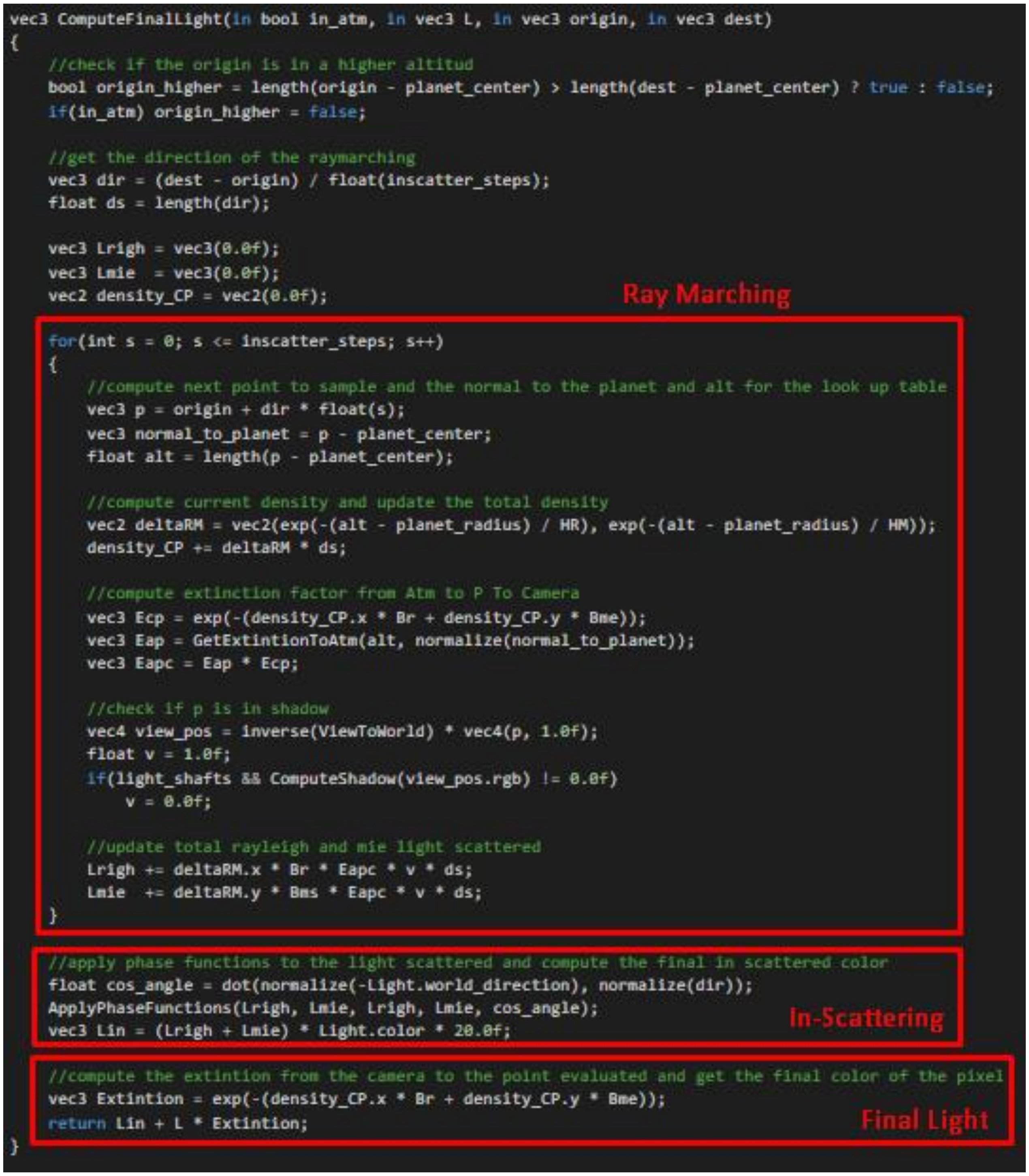
Now that we have L_O we need to compute the end points and provide all that information to the function that performs the raymarching in order to get the final light result:



```
   //check if the fragment is the space itself
   bool point_in_space  = length(normal_planet) > (atm_radius - radius_epsilon);
   bool cam_in_space    = length(camPos - planet_center) > atm_radius - radius_epsilon;

   //compute end points of the ray to ray march for the inscatter light
   vec3 Origin = vec3(0.0f);
   vec3 Dest = vec3(0.0f);
   bool intersect = ComputeEndPoints(point_in_space, cam_in_space, world_pos, Origin, Dest);

   //check if not intersection or the ray collides with an object before reaching atmosphere
   if(!intersect || length(camPos - Origin) > length(camPos - world_pos) && cam_in_space && point_in_space)
   {
        vec3 result = diffuse + specular;
        if(Z == 0.0f) result += SpaceColor;
        FragColor = vec4(result, 1.0);
        return;
   }

   //compute the atmoshperic scatte color and apply the space color if needed
   vec3 result = ComputeFinalLight(cam_in_space && point_in_space, Lground, Origin, Dest);
   if(Z == 0.0f) result += SpaceColor;

   //set color Normal
   FragColor = vec4(result, 1.0);
}
```

Figure 10. Compute endpoints and get the final light of the current pixel.

To compute the final color, we will apply a simple ray marching over the path delimited by the two end points accumulating the total Rayleigh and Mie scattered light, once all the samples are computed we will apply the phase functions to the Rayleigh and Mie scattered lights and then compute the in-scattering light to finally compute the final light color of the point evaluated.



```
vec3 ComputeFinalLight(in bool in_atm, in vec3 L, in vec3 origin, in vec3 dest)
{
    //check if the origin is in a higher altitud
    bool origin_higher = length(origin - planet_center) > length(dest - planet_center) ? true : false;
    if(in_atm) origin_higher = false;

    //get the direction of the raymarching
    vec3 dir = (dest - origin) / float(inscatter_steps);
    float ds = length(dir);

    vec3 Lrigh = vec3(0.0f);
    vec3 Lmie  = vec3(0.0f);
    vec2 density_CP = vec2(0.0f);

    for(int s = 0; s <= inscatter_steps; s++)
    {
        //compute next point to sample and the normal to the planet and alt for the look up table
        vec3 p = origin + dir * float(s);
        vec3 normal_to_planet = p - planet_center;
        float alt = length(p - planet_center);

        //compute current density and update the total density
        vec2 deltaRM = vec2(exp(-(alt - planet_radius) / HR), exp(-(alt - planet_radius) / HM));
        density_CP += deltaRM * ds;

        //compute extinction factor from Atm to P To Camera
        vec3 Ecp = exp(-(density_CP.x * Br + density_CP.y * Bme));
        vec3 Eap = GetExtintionToAtm(alt, normalize(normal_to_planet));
        vec3 Eapc = Eap * Ecp;

        //check if p is in shadow
        vec4 view_pos = inverse(ViewToWorld) * vec4(p, 1.0f);
        float v = 1.0f;
        if(light_shafts && ComputeShadow(view_pos.rgb) != 0.0f)
            v = 0.0f;

        //update total rayleigh and mie light scattered
        Lrigh += deltaRM.x * Br * Eapc * v * ds;
        Lmie  += deltaRM.y * Bms * Eapc * v * ds;
    }
    //apply phase functions to the light scattered and compute the final in scattered color
    float cos_angle = dot(normalize(-Light.world_direction), normalize(dir));
    ApplyPhaseFunctions(Lrigh, Lmie, Lrigh, Lmie, cos_angle);
    vec3 Lin = (Lrigh + Lmie) * Light.color * 20.0f;

    //compute the extintion from the camera to the point evaluated and get the final color of the pixel
    vec3 Extintion = exp(-(density_CP.x * Br + density_CP.y * Bme));
    return Lin + L * Extintion;
}
```

Figure 11. Function that perform the ray marching to compute the final light of a point.

3.4 IMPORTANT DETAILS

- **Cascade Shadow Maps:** It is not mandatory to implement this shadow system but is something that must be considered since the magnitude of the scene that are usually represented by this kind of atmospheric effect are usually huge. This implementation is the perfect fit for the scene rendered.
- **Reversed Z depth:** Since we are representing actual planets the values that the depth buffer can get are very large, this will produce a lot of artifacts for objects that are very far from the camera. One way to solve this problem is to reverse the Z depth buffer, so we will achieve a linear precision for every point in the scene.

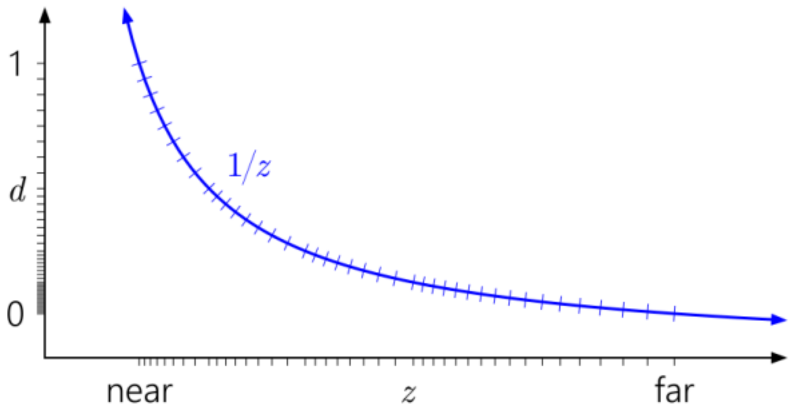
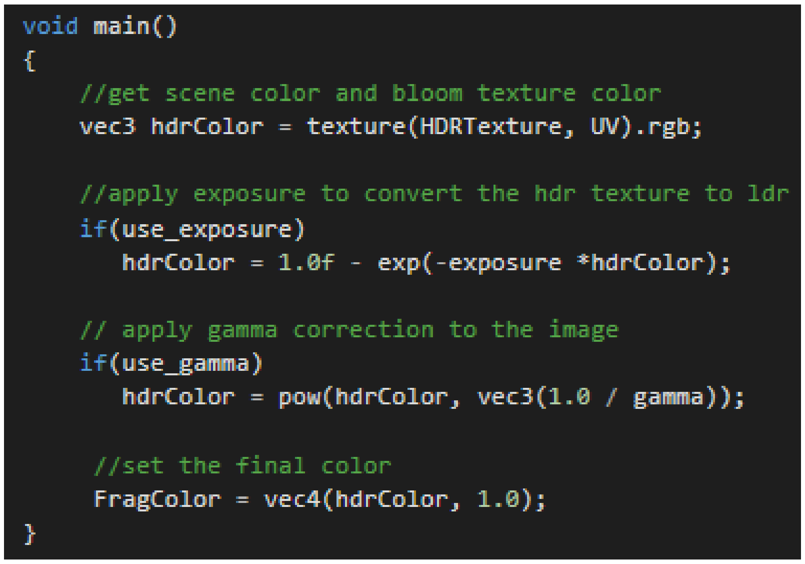


Figure 12. Reversed Z depth buffer precision graph.

- **High Dynamic Range:** Since we are working with a lot of colors that get greater than the normal range of [0,1] we need to implement HDR to get a realistic result.



```
void main()
{
    //get scene color and bloom texture color
    vec3 hdrColor = texture(HDRTexture, UV).rgb;

    //apply exposure to convert the hdr texture to ldr
    if(use_exposure)
       hdrColor = 1.0f - exp(-exposure *hdrColor);

    // apply gamma correction to the image
    if(use_gamma)
       hdrColor = pow(hdrColor, vec3(1.0 / gamma));

     //set the final color
     FragColor = vec4(hdrColor, 1.0);
}
```

Figure 13. Shader that implements HDR.

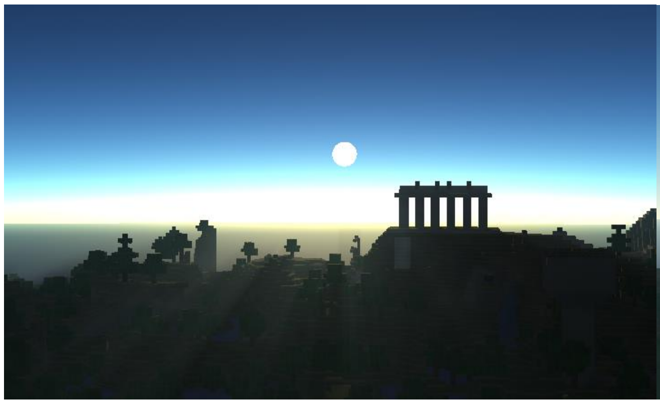
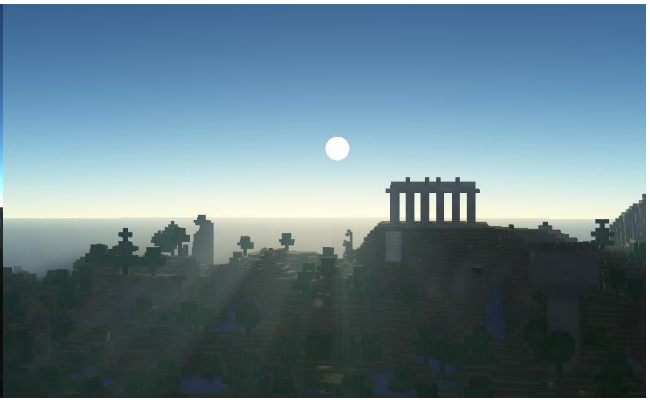
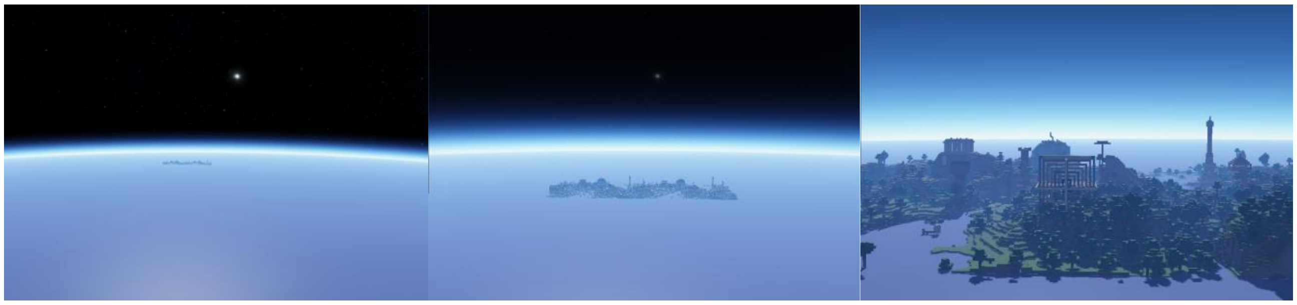



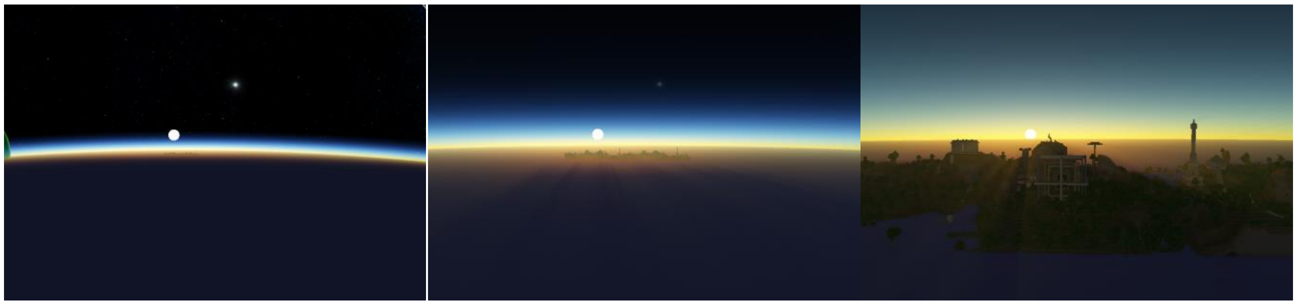
Figure 14. Not using HDR.

Figure 15. Using HDR.

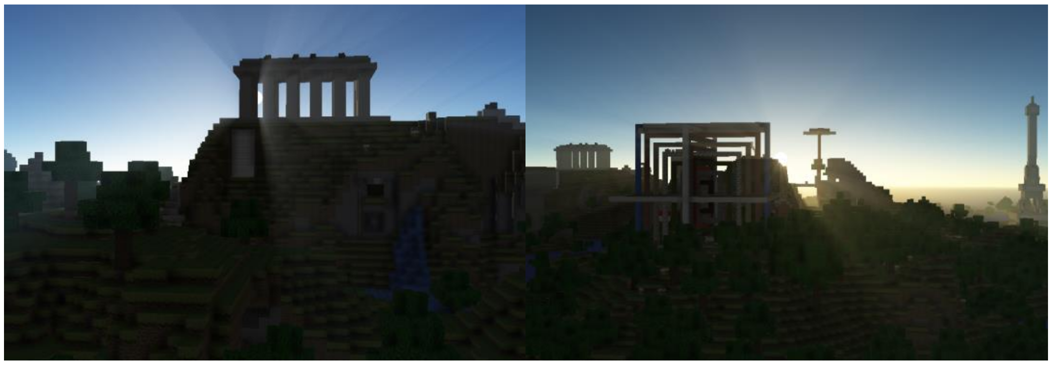
4. RESULTS

Smooth transitions between space and planet surface





Light Shafts



5. IMPROVEMENTS

- **Precompute In-Scattering Light:** This approach tries to precompute the in-scattering light in any altitude and any view direction and any sun direction. Therefore, for this approach we need to use a 2D_ARRAY_TEXTURE or a 3D_TEXTURE to store all the different information. The performance gain will be considerable, but at the same time a huge new amount of memory will be sent to the GPU so we will sacrifice memory for speed. One big problem of this approach is that the light shafts will need to be handled separately since the precomputation is made assuming a perfect sphere as a planet, so shadows cannot be considered. Depending on the resolution of the ARRAY there will be some artifacts when passing from one texture to another.

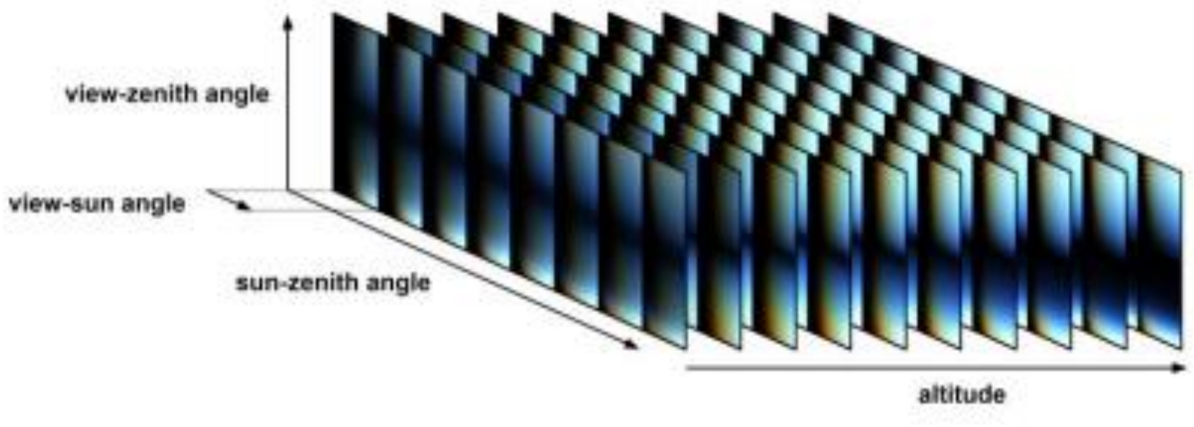


Figure 16. 2D_ARRAY_TEXTURE storing the precomputation of the in-scattering.

- **Epipolar-Sampling:** This approach tries to reduce the number of raymarching that we do, for the current approach we are performing raymarching for each pixel in the screen that collides with the atmosphere. With epipolar sampling the purpose is to perform raymarching only on the epipolar lines of the screen. And performing some tricks like checking for depth discontinuities to add more samples and interpolate between samples in the same epipolar lines we can achieve a very good result gaining a lot of performance. The biggest problem of this approach is that is a bit complex to implement, but if you want to implement realistic atmospheric scattering for a real time application it would be a worth try.

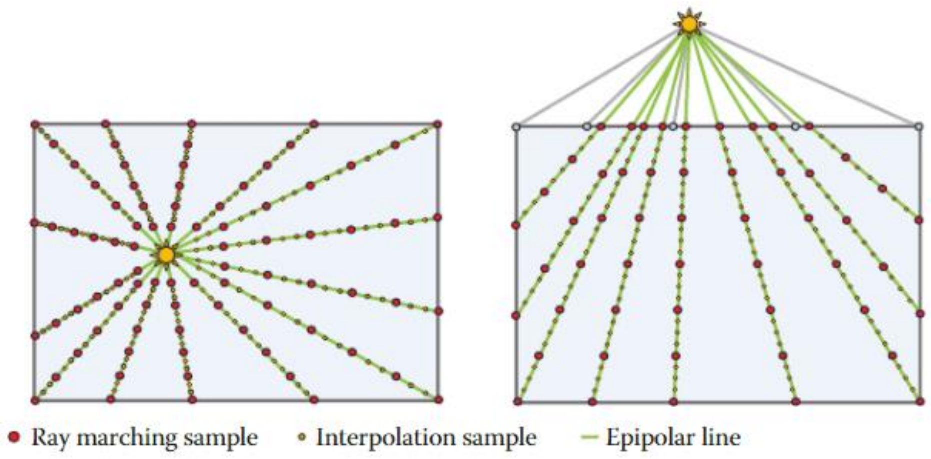


Figure 17. Epipolar sampling example.

- **Clouds:** Implementing clouds on the scene would be a good idea since it will be applied as a Mie scattering effect and will produce more realistic results in the final image.

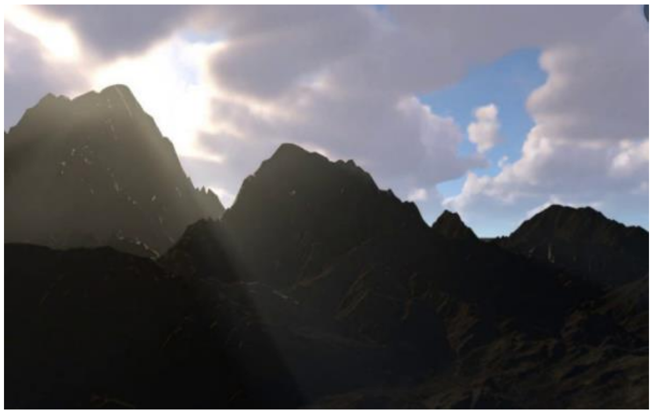


Figure 18. Cloud simulation example.

- **Lens Flare:** This would be a very good post process effect to consider implementing since it gives a nice result without sacrificing performance and is one of the best scenarios to implement this effect for the sun.

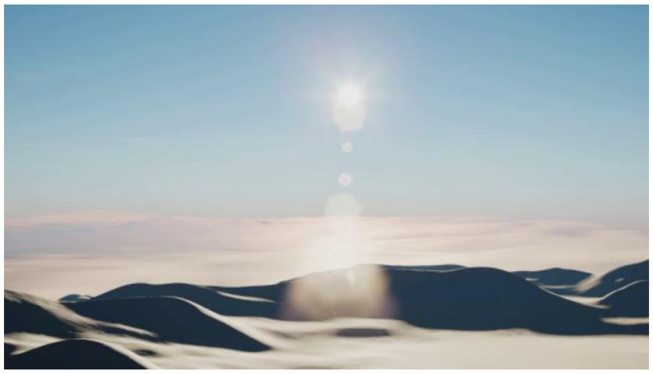


Figure 19. Lens Flare example.

6. CONCLUSIONS

The simulation runs in real time giving very good results, the difference isn't noticeable from 30 integration steps, the problem comes when we render the light shafts, that doesn't get very realistic because there isn't enough precision. If we increase the steps, we will get a much more realistic result, but the performance drop will be huge. In conclusion we should think of a way to handle the light shafts separately and then add it to the in-scattered in order to obtain a more realistic result.

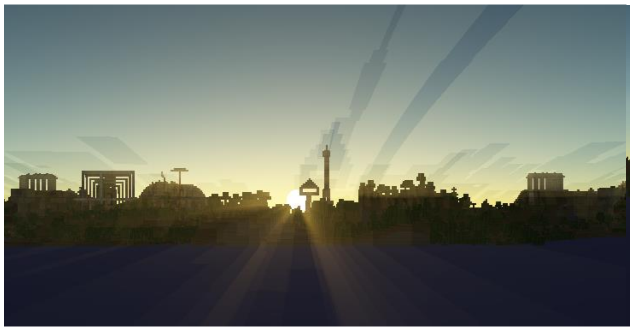


Figure 20. Real time, 20 steps.

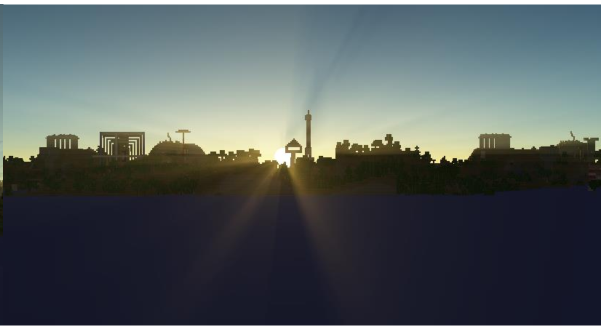


Figure 21. Not real time, 800 steps.

This technique can be implemented with a lot of different techniques in order to make the scene look more realistic, like Ambient Occlusion, Skybox, Cascade Shadow Maps, etc.

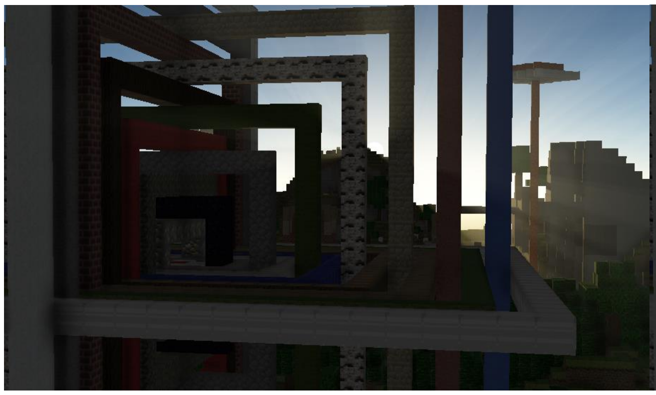


Figure 22. Horizontal Ambient Occlusion.

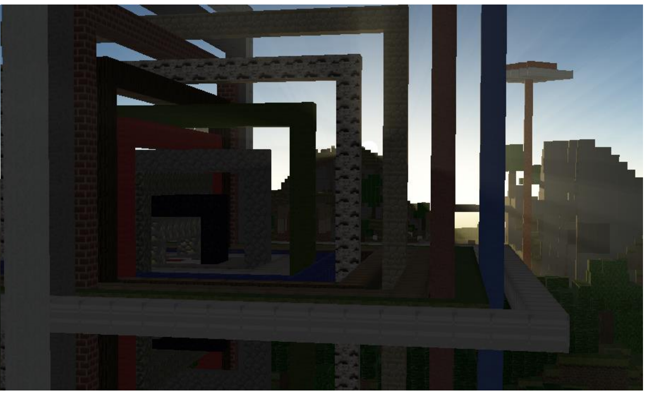


Figure 23. Not using Ambient Occlusion.

To sum up, I think that the technique is very interesting, it isn't very complex once you get into it and the results are very good. An atmosphere is something that almost any outdoor game has so it is very nice to implement something like this.

7. BIBLIOGRAPHY

Bodare, G., Sandberg, E. (2014). *Efficient and Dynamic Atmospheric Scattering* [Master of Science Thesis in Computer Science – Algorithms, Languages and Logic, Chalmers University of Technology]. https://www.chalmers.se/en/Pages/default.aspx

O'Neil, S. (2005). *Chapter 16. Accurate Atmospheric Scattering* in Fernando, R. Editor (Ed.) GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Chapter 16). Addison-Wesley Professional.

Sperlhofer, S. (2011). *Deferred Rendering of Planetary Terrains with Accurate Atmospheres* [Master's Thesis, University of Applied Sciences Technikum Wien Game Engineering and Simulation]. https://www.technikum-wien.at/en/

Yusov, E. (2014). *High Performance Outdoor Light Scattering Using Epipolar Samling* in Wolfgang, E. Editor (Ed.) Advanced rendering Techniques (pp. 101-126). CRC Press Taylor & Francis Group, LLC.