Project Report - Real-time Volumetric Cloud Rendering

Supervised by Amal Dev Parakkat and Kiwon Um

Philippe Telo

IGR202 - Informatique graphique et réalité virtuelle

TELECOM
Paris

Abstract

Volumetric rendering produced stunning visual effects in the film industry for quite a long time. But it has only been a few years since the consumer hardware was powerful enough to use the technique in real time. The game Horizon Zero Dawn is the first one that featured real-time volumetric clouds. Here I propose an implementation of a ray-marching algorithm and a cloud density map generator, that runs in real time.

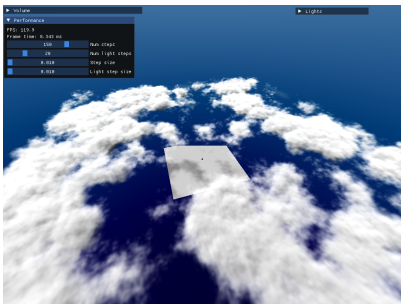


Figure 2: Little sneak-peak of the result

Contents

1 Literature Survey

1.1 State of the art

In the field of real time volumetric clouds rendering, one project that revolutionized the game was the video game Horizon Zero Dawn ([3]), as they were the first to use volumetric clouds in a game, in real-time. It was very hard to achieve this because ray-tracing, and especially volume raytracing, is very computationally heavy, and is traditionally limited to offline rendering. They used clever tricks and optimisations to get stunning clouds in very low render time.

Figure 3: Enter Caption

To help me understand better the subject, I also used [2] and [1].

In this section, I will explain briefly the paper I based my project on, without going too deep into the details. Do not worry, we will get more technical in 2.

This paper is roughly composed of two parts: the cloud modeling (1.2), and the volume rendering (1.3). Everything is thought out for real-time, and the author explains in details the process that led to these results.

1.2 Cloud modeling

1.2.1 Base shape

The base shape of the cloud is obtained by multiplying a series of signals:

- A base low resolution 3D noise map, used by adding several noise function with increasing frequency and decreasing amplitude, as explained in 2.1.2. To speed up the computations, the author use tiling, pre-computed noise textures.
- A finer 3D details map, that will be substracted from the base map.

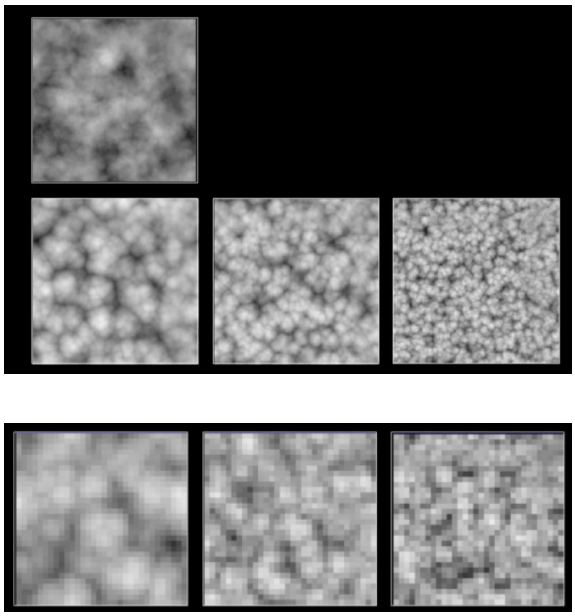
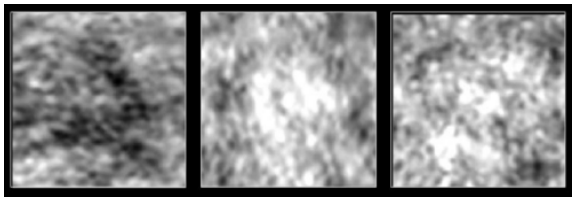
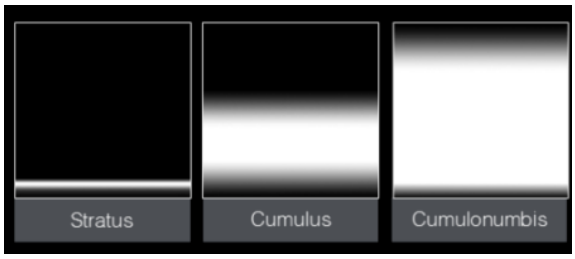


Figure 4: The Voronoi noise textures, stored in the three channels of a texture

- A 2D curl noise texture, that will help fake the turbulence of the clouds:



- Finally, they weight this with height gradients, corresponding to the different cloud types:



1.3 Volume rendering

For the volume rendering, the author use an algorithm called **ray-marching**. Instead of the traditional triangle-based rendering, this technique simulate the rays of light, going outward from the camera, and traversing the volume.

First, a primary ray is shot from the camera to determine the opacity of the cloud. To achieve that, we march the ray using a certain step size, and accumulate the density we sample at each step. The final density is an approximation of the density along the cloud, and are used to compute the opacity using Beer's law (1).

$$Energy = e^{-d*a} \tag{1}$$

With a a factor that control how much the cloud absorbs light, and d the total density along the ray. The Beer's Law can also represent the probability a ray hits a particle inside the cloud, based on density and distance.

Here is a schematic representing how the primary ray is shot:

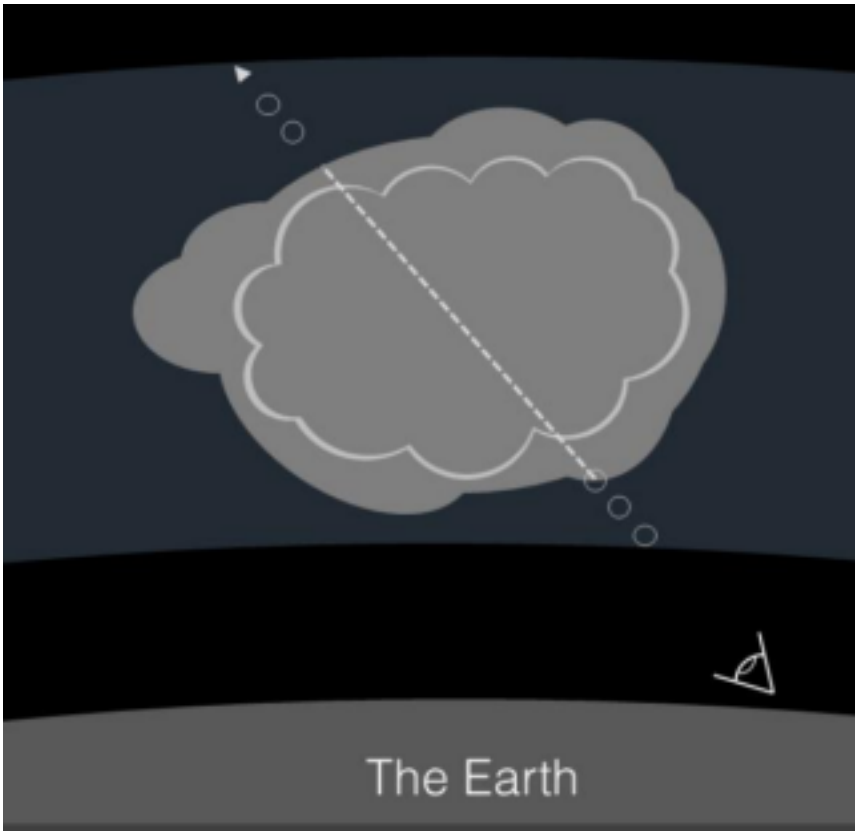


Figure 5: A single primary ray

Now, to obtain a perfect result for the cloud lighting, we theoretically have to get the light contribution from all directions at each point of the cloud. But we cannot afford that, so the simplest way to approximate this in-scattering (the amount of light scattered towards the eye), as proposed by the author, is to shot a secondary ray from the point in the cloud toward the light source, and scale it by a phase function (a function depending on the dot product between the ray direction and the light direction).

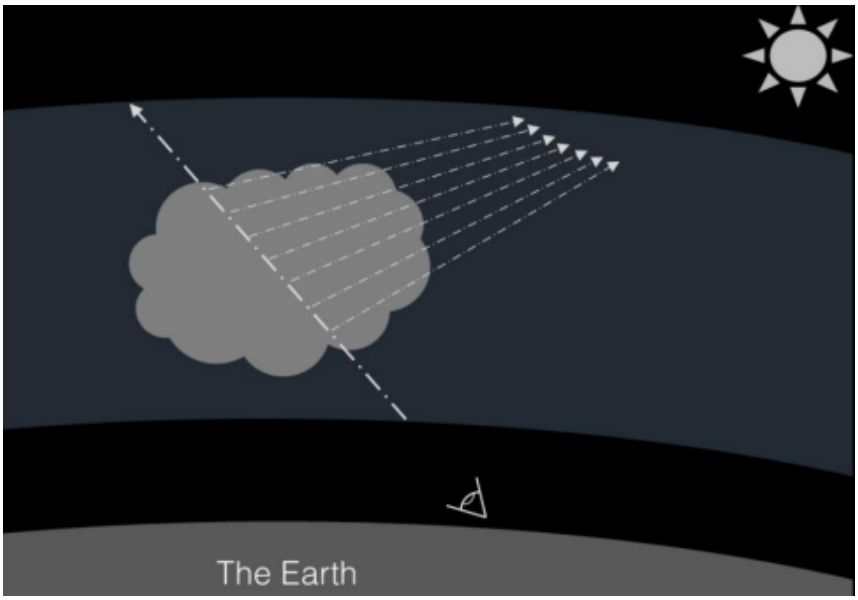


Figure 6: Secondary light rays

We will get into more details on how the final radiance is computed in 2.3.

2 The algorithm

I will present here how I implemented the algorithm, along with pictures and code pieces.

2.1 Cloud generation

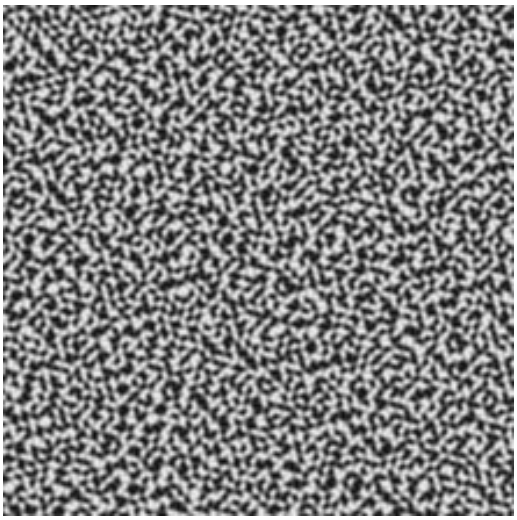
First, we need a way to represent the cloud in memory. We will use a density map, that will be sampled by the ray-marching algorithm for the rendering. We have two limitations

- Each point of the density shall be computed independently. That way, we can parallelize the computations to speed up the process.
- The density computations shall stay very simple, because we need to compute them in real time, and for each voxel of the 3D texture.

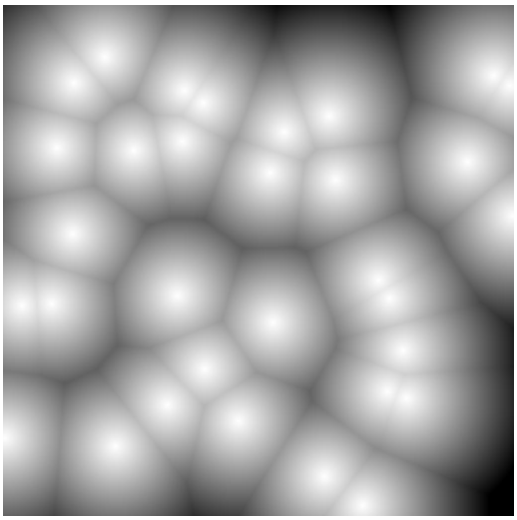
We obtain a very simple algorithm: for each pixel p of a WxHxD texture, get the value $f(p)$ from the density function and write it in the texture. This way, we can easily use multi-threading or compute shaders to generate this texture really fast.

2.1.1 Noise functions

To generate random clouds that are still coherent (white noise is "too" random), we can use coherent noise. These are functions that are random, but spatially coherent (if x and y are close, f(x) and f(y) are also close). For this project, I will use simplex noise for this project, which is an upgrade of the Perlin noise algorithm created by Ken Perlin while working on Tron. The implementation of this algorithm is outside the scope of this project, so I will use an existing implementation. Voronoi noise, which give very organic textures, is also a very good option for clouds, but harder to implement in a compute shader, so I will not be using it.



(a) Simplex noise function (we will use this one)



(b) Inverted Worley (Voronoi) noise function

Figure 7: Noise functions, sampled on a 2D plane

2.1.2 Fractal brownian motion

Using simplex noise, the randomness is coherent, but all the features of the cloud are at the same scale. It is enough for some sort of fog for example, but if we want well structured shapes, we can use what is called fractal brownian motion. The idea is to sum noise functions with increasing frequency and decreasing amplitude. This way, we have well defined low frequency shapes, and high frequency details. There is an example code for this function:

```
float fbm(vec3 pos, int octaves) {
    float noiseSum = 0.0;
    float frequency = 1.0, amplitude = 1.0;
    for(int i = 0; i < octaves; ++i) {
        noiseSum += snoise(pos * frequency + i * offset) * amplitude; // Offset to
        amplitude *= 0.7; // arbitrary (< 1)
        frequency *= 2.58; // arbitrary(> 1)
    }
    return noiseSum;
}
```

We can visualize the way it works like this (from `https://www.scratchapixel.com/`):

If we extend this to 3D, we can get nice structured clouds.

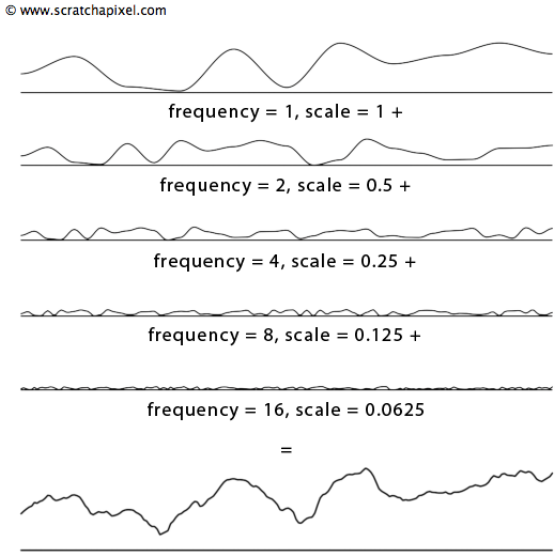


Figure 8: Sum of noise functions

2.1.3 Using these to generate the cloud shape

Finally, we have the necessary functions to generate the density map that will give us good looking clouds. To achieve this, I went step by step to go closer and closer convincing clouds. Note that this is a purely arbitrary method, that I fine-tuned to make the clouds look like what I wanted.

The base coverage value is given by a simple sample of the snoise function in 2 space. We get a very smooth 2D texture, that will define the base shape of the clouds.

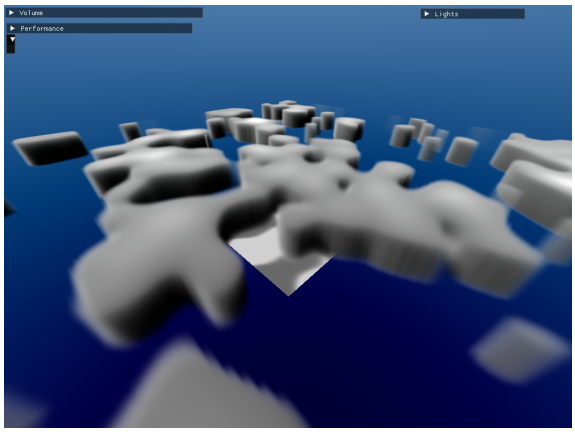


Figure 9: Base coverage

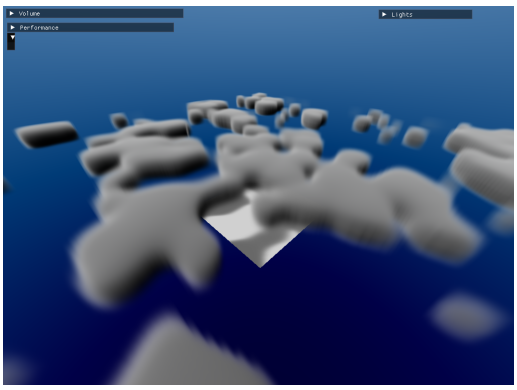
I then used a function that returns a factor based on the local height inside the

cloud, to round the shape a bit:

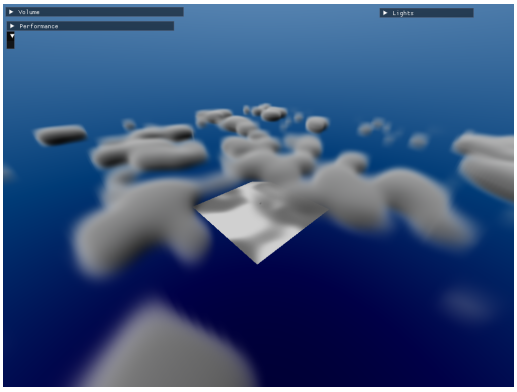
$$\begin{array}{rccc} heightFactor & : & [-1,1] & \rightarrow & [0,1] \\ & & h & \mapsto & heightFactor(h) = 1 - |h| \end{array}$$

We also pow the resulting density to a certain exponent α, to round the shape even more.

Which gives us:



(a) $\alpha = 1$



(b) $\alpha = 1.5$

Figure 10: Rounded coverage map

Finally, we sample a details value from the snoise function, at a greater frequency and in 3D this time. We then substract the details from the coverage map (and min the result with zero to avoid negative densities). We obtain this:

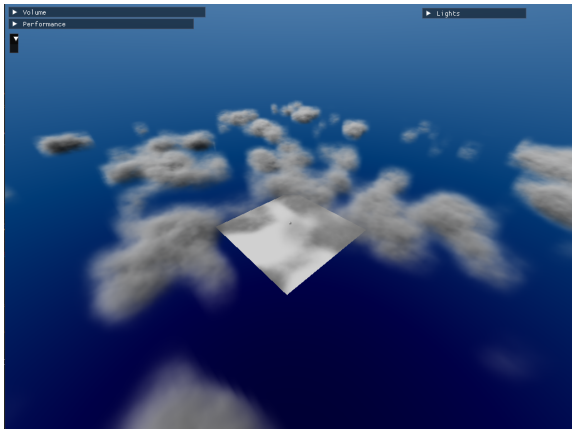


Figure 11: Adding a details map

After some tweakings, here is the final iteration of the density texture generation algorithm:

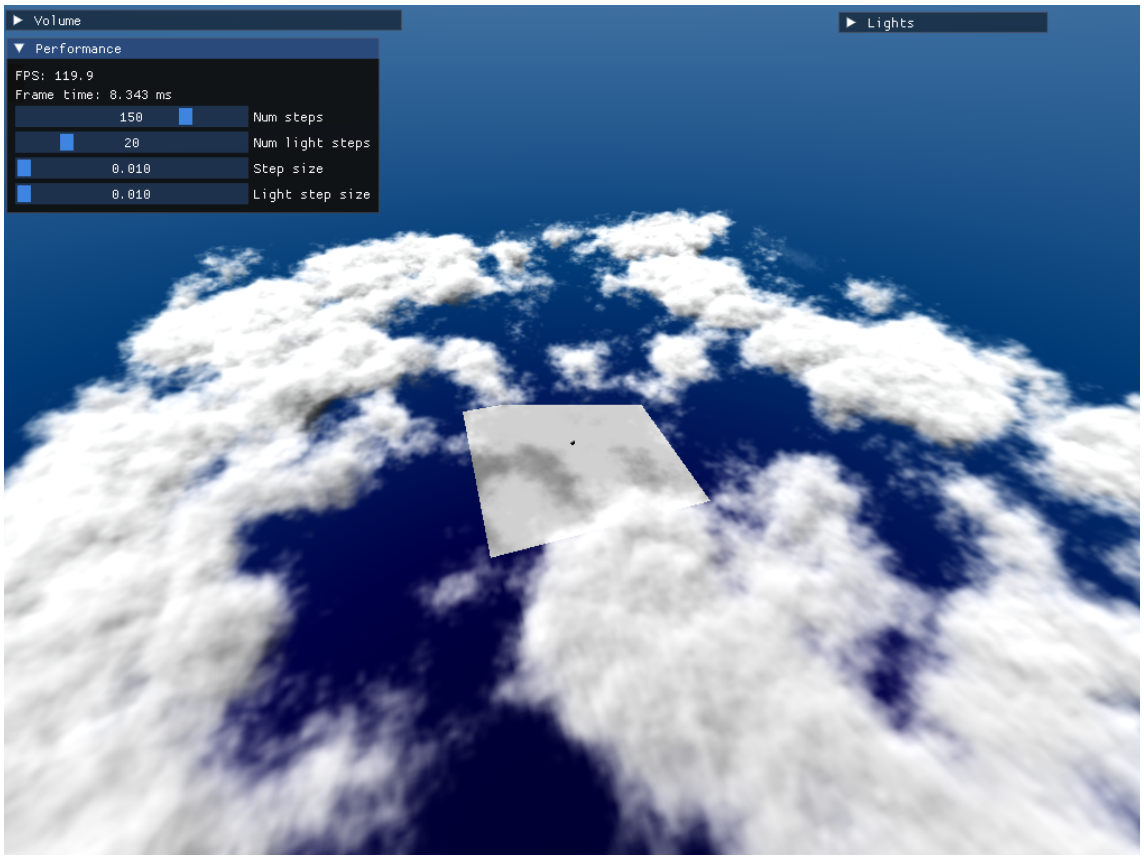


Figure 12: Final results, view from above

2.2 Pipeline

For the implementation, I ended up using a deferred rendering pipeline, to be able to use traditional rendering for solid objects, and raymarching for the clouds, and use the best of the two methods.

Thus, there are three passes each frame:

- **Density pass**: the density map is rendered in a 3D texture using a compute shader, to fully exploit the parallel capabilities of the GPU
- **Geometry pass**: traditional rasterizing to render all the solid objects, but we use a frame buffer object to write the albedo, normal and position data in the G-buffer, for use by the final pass.
- **Lighting + raymarching pass**: Finally, we use the albedo, normal and position data to compute the final surface color, and do the raytracing calculations on top, to get the final render color.

Using this pipeline, we can independently program the density map generation, the solid objects rendering, and the clouds rendering, and optimize each stage more easily. It also enable us to make some improvement, as proposed in 4.2.

2.3 Raytracing

To render the volume, we use the ray-marching algorithm we talked about in 1.3. The goal is to compute for each pixel the render color, based on the cloud transmittance, the light scattered from the cloud, the sky color and the color of the solid objects.

First, we have to compute the view direction from the fragment coordinate on screen, so we can shoot a ray in this direction. With a series of transformations using the view and projection matrices, we get *rayDir* and *rayPos*:

```
// uv are the screen coordinates normalised in [0, 1]
vec4 clip = vec4(uv, -1.0, 1.0); // Screen space
vec4 eye = vec4(vec2(u_invProjMat * clip), -1.0, 0.0); // View space
vec3 rayDir = normalize(vec3(u_invViewMat * eye)); // World space
vec3 rayOrigin = u_invViewMat[3].xyz; // Little trick to get the camera pos
//from the view matrix
```

We now have the ray origin and direction, we can start the ray-tracing calculations! First, we define a cloud domain, an area outside of which the density will be zero. A rectangular box, to make the calculations easier. We can then project the ray onto the surface of this box, to skip all the empty space before, and start directly in the cloud.

2.3.1 Cloud traversing (primary ray)

```
vec4 raymarchCloud(vec3 rayOrigin, vec3 rayDir, float trender) {
    float transmittance = 1.0; // 1 - Opacity
    vec3 lightEnergy = vec3(0); // Light scattered

    for(int i = 0; i < MAX_STEPS && t < tmax; i++) {
        vec3 p = rayOrigin + rayDir * t;
        float density = sampleDensity(p);

        for(int j = 0; j < u_numLights; j++) {
            float lightTransmittance = lightMarch(p, u_lights[j]);
            float phase = phase(dot(rayDir, rayDir));
            lightEnergy += density * stepSize * transmittance * lightTransmittance
        }
        transmittance *= exp(-density * stepSize * u_cloudAbsorption);
```

```
        t += stepSize;
    }
    return vec4(lightEnergy, transmittance);
}
```

2.3.2 Cloud traversing (secondary/light ray)

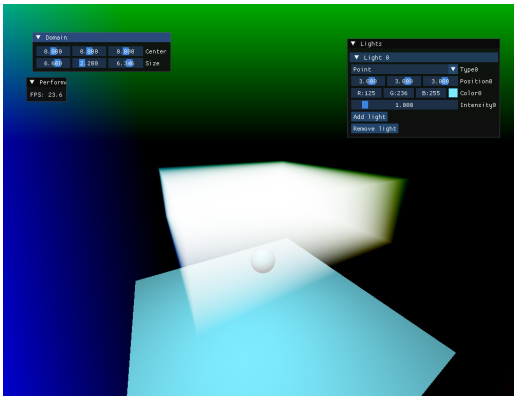
```
float lightMarch(vec3 rayOrigin, Light light) {
    vec3 lightDir = normalize(light.direction);
    float totalDensity = 0;
    float t = 0;

    for(int i = 0; i < MAX_LIGHT_STEPS; i++) {
        vec3 p = rayOrigin + lightDir * t;
        float d = sampleDensity(p);
        totalDensity += d * stepSize;
        t += stepSize;
    }

    return exp(-totalDensity * u_lightAbsorption);
}
```

We can finally compute the render color:

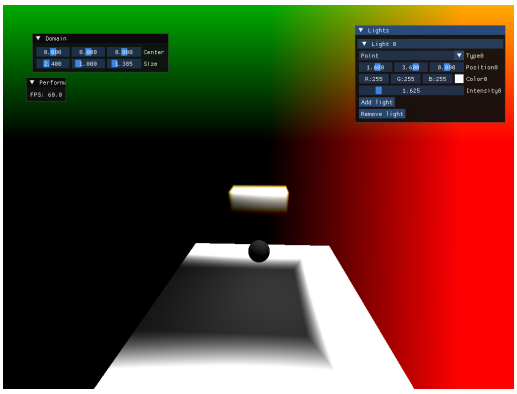
```
vec3 renderColor = solidColor * transmittance + lightEnergy;
// solidColor is the color given by the geometry pass
```

3 Results

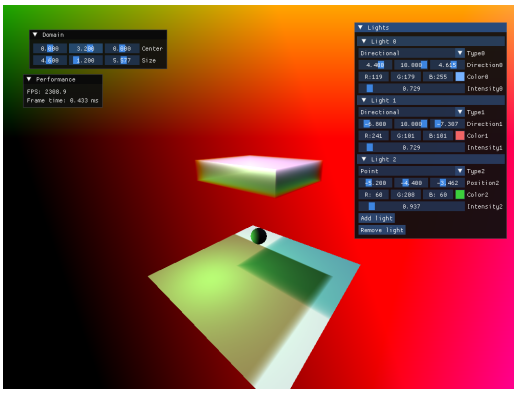
Here are the step by step results that I got.



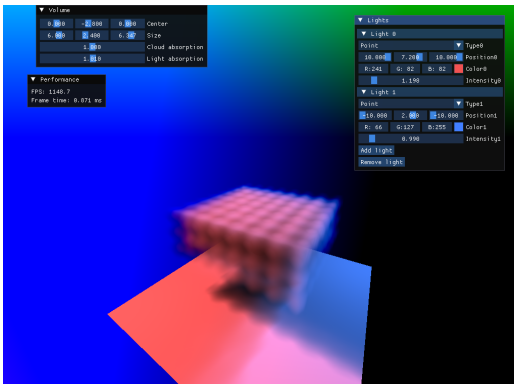
(a) Constant density, no lighting



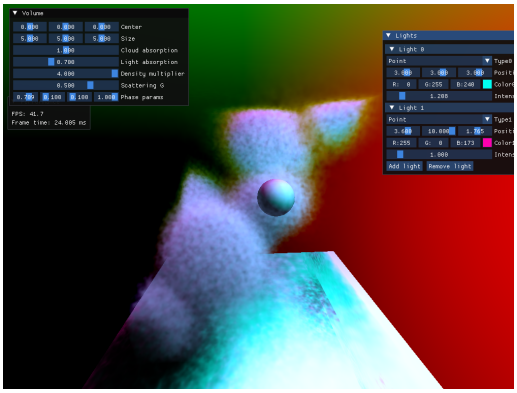
(b) Added light ray + shadows on solid geometry



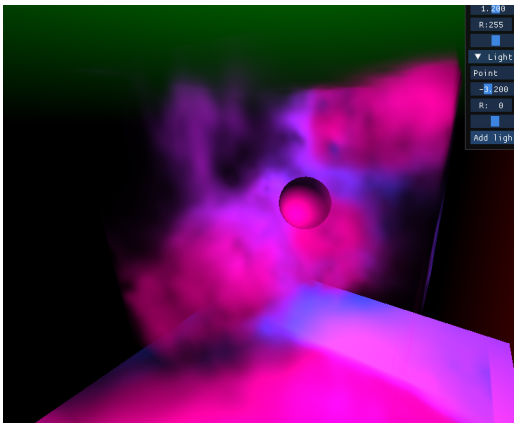
(a) Constant density, multiple colored lights



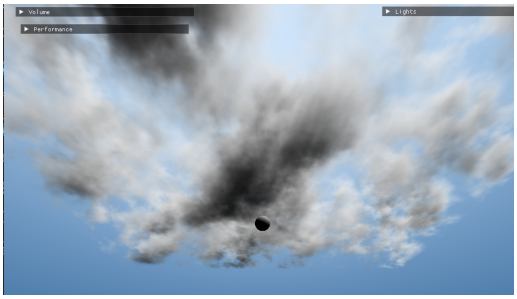
(b) More complex hardcoded density function



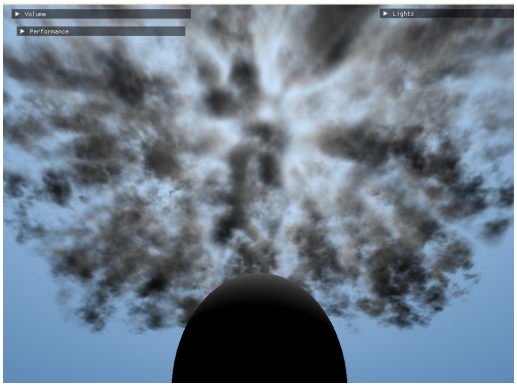
(a) First success with a compute shader and a texture



(b) Improved the generation algorithm



(a) Large scale density map...



(b) ... and a blue sky!

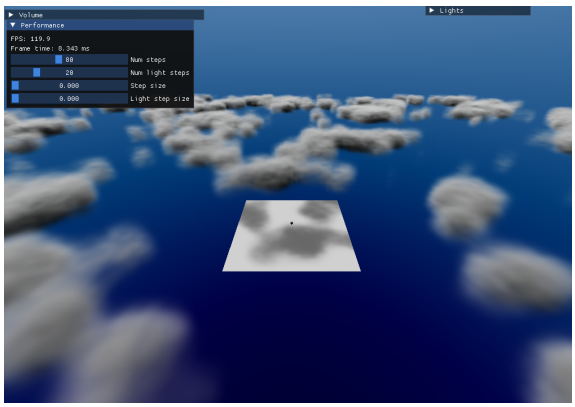


Figure 17: Another density function

3.1 Final results

This is the final density map I came up with. It uses a base 2D coverage map, to which I substracted a higher frequency details map, as explained in 2.1.3. I also tweaked the rendering parameters to make the most realistic clouds I could. Here with the rendering parameters: $density\_max = 2$, $lightAbsorption = 0.3$ and $forwardScattering = 0.74$

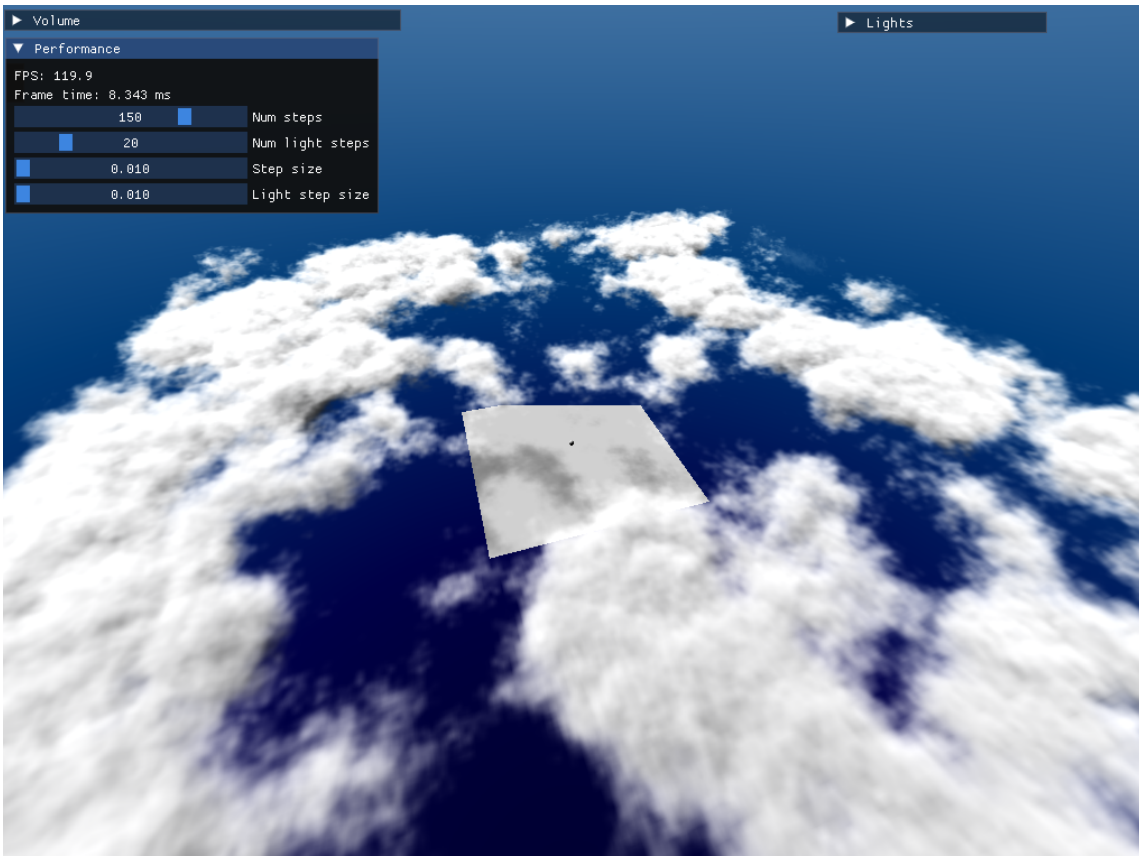


Figure 18: Final results, view from above

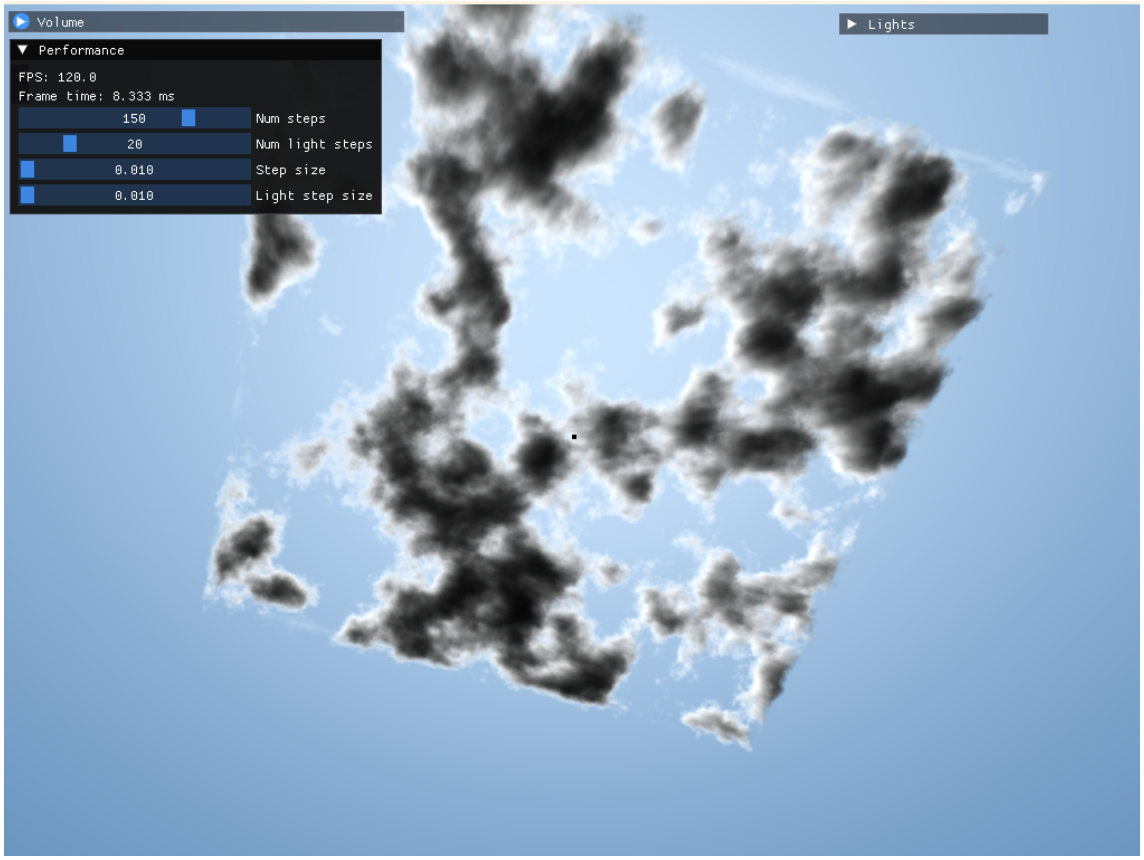


Figure 19: Final results, view from below

Here are two real-world pictures, to compare.

(a) From above (Source: wikimedias)

(b) and below (Source: DeviantArt)

We can see that in our implementation, the light is absorbed more by the volume, so our clouds look darker from below than the reference pictures. This can be easily

fixed by tweaking the lightAbsorption and phase parameters.

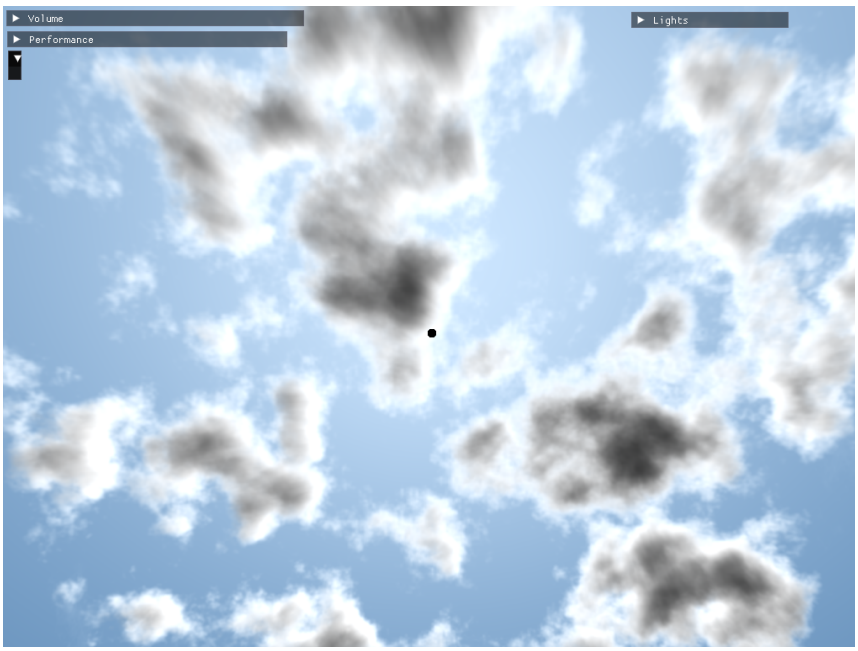


Figure 21: Final result, with $density\_max = 1$, $lightAbsorption = 0.25$ and $forwardScattering = 0.75$

We can see the caracteristic silver lining of our simulated clouds, and visualise their depth well, with the variation in brightness.

I we look closely, we can also see some god rays, which makes me quite happy.

(a)

(b)

4 Improvements and ideas

4.1 Improvements

For this project, I chose to use a deferred pipeline. This way, I can add this renderer on top on any existing project using a traditional pipeline or a deferred one. I might change the texture requirement to a depth texture instead of a positions texture, as it is more widely used, and directly supported by OpenGL.

Since we have the world space position of the geometry in the lighting shader, we can shoot a light ray from there to determine the shadow caused by the cloud. This way, we get realistic shadow without coding anything else. The only downside is that the solid objects don't cast shadows. We could use traditional shadow mapping to fix this, but I didn't implement it in this project.

4.2 Ideas

During the development of this project, I had several ideas to improve it. As we already are using deferred rendering, we could separate the lighting pass from the cloud rendering pass, and compose them in a final pass. This way, we can easily change the frame buffer to render the clouds at only quarter resolution (width/2 and height/2), and upscale the result. Because the clouds have quite low frequency details, and are very heavy to computes, this will be barely noticeable and will speed up the rendering a lot.

In the current state of thing, the step size is constant during all the cloud traversal process. This could be improved, as low density have a very low contribution to the final color, and high density zone contribute much more, but the max absorption is quickly reached - the processing power is wasted on low density zones, and they can be quite large. If we could somehow have larger steps in low density zones, and shortest steps in high density ones, we would save some processing power, and increase the final quality. We could achieve that by starting with large steps while traversing empty space, and as soon as we hit a cloud ($density(x) > threshold$), we roll back one step and decrease the step size.

Finally, I would like to make a small flight simulator out of this project. I can regenerate the density map each frame thanks to the speed of compute shader, and because the raster pipeline and the raytracing pipeline are separated, it would be easy to integrate a game in it. Using another compute shader, I could even generate some terrain in real time to make an interesting landscape.

References

[1] Fredrik Haggström. "Real-time rendering of volumetric clouds". In: *Degree Project in Computing Science Engineering* (2018). URL: https://www.diva-portal.org/smash/get/diva2:1223894/FULLTEXT01.pdf.

[2] y Patapom / Bomb! "Real-Time Volumetric Rendering". In: *Revision Demo Party* (2013). URL: https://patapom.com/topics/Revision2013/Revision%202013%20-%20Real-time%20Volumetric%20Rendering%20Course%20Notes.pdf.

[3] Andrew SCHNEIDER. "THE REAL-TIME VOLUMETRIC CLOUDSCAPES OF HORIZON ZERO DAWN". In: *SIGGRAPH* (2015). URL: https://www.guerrilla-games.com/media/News/Files/The-Real-time-Volumetric-Cloudscapes-of-Horizon-Zero-Dawn.pdf.