Real-Time Volumetric Rendering

By Patapom / Bomb! *(spring 2013, course notes for the Revision Demo Party)*

Almost everyone in the demoscene is enjoying ray-marching through a distance field but I haven't seen much volumetric rendering done with it.

So, why not use ray-marching to achieve what it was created for originally: rendering a participating medium?

What's a participating medium?

- ⇨ It's a volume (the medium) where refraction, density and/or albedo changes locally

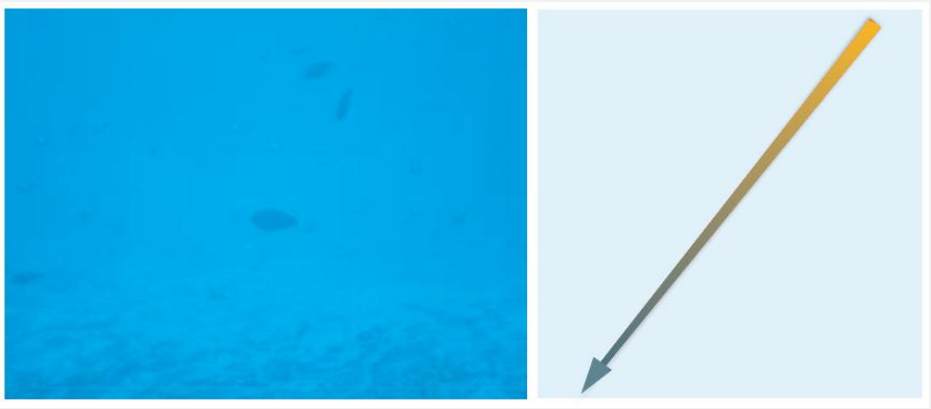
All medium is in fact a participating medium at some level; it all depends on the distances we're considering. But some media can be thought of as "homogeneous" on short distances.

Homogeneous Medium (air, water, glass) on short distances:

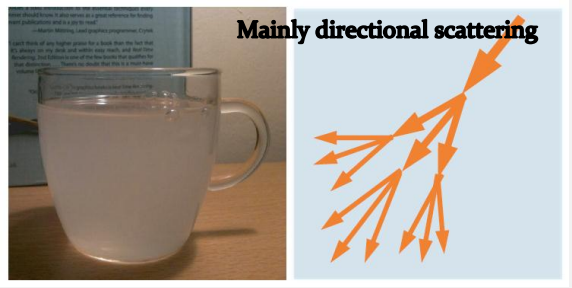
In most cases though, we have to consider a heterogeneous medium with characteristics that vary more or less quickly.

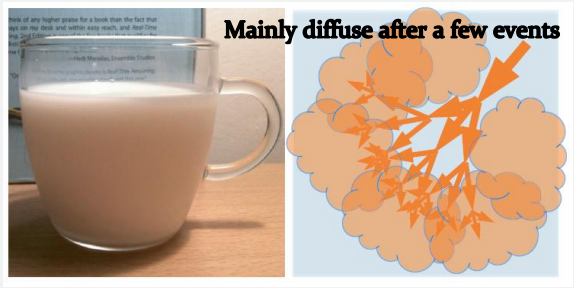
What can happen to a flux of photons when we make these qualities change?

Absorption, which may change according to wavelength, like we see here: a blue coloring occurs after long distances in water.

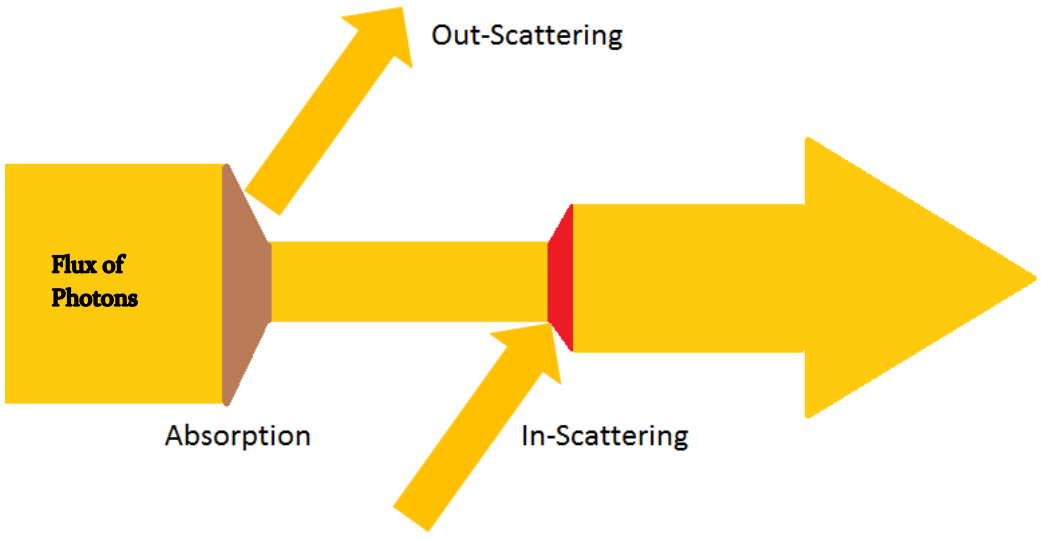


Scattering; photons are bouncing off of particles in various directions.
Depending on the density and type of particles, scattering is more or less random, more or less frequent.





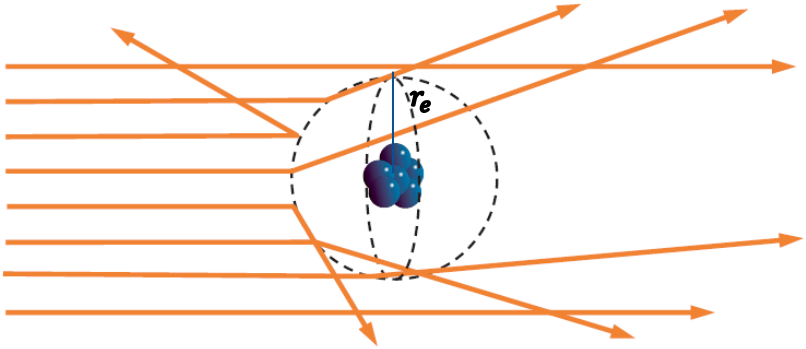
To sum up, what happens to a flux of photons is one of these events:



(I'll come back later to a last type of event: emission)

Scattering & Extinction

Consider a single particle (molecule, dust, a water droplet or whatever):



Photons are going to interact with it and bounce off of its "surface".
In the case of clouds, imagine a single microscopic water droplet: the droplet will either refract or reflect the rays of light, but almost none of them will be absorbed; this is why clouds are so bright: their albedo (ratio of reflected over incoming light) is nearly 100%.

What happens if light is traveling in a volume filled with a large density of such particles?

In the end, it's a game of chance: what is the chance I hit a particle after N meters?

Particle Cross-Section $a = \pi {r_e}^2 \quad (m^2)$

r_e is the effective radius (typical radius for water droplets in a cloud $\sim 2 - 20\mu m$)

Absorption/Scattering Cross-Section $\sigma = a\, N_0 \quad (m^{-1}) \qquad (\mathbf{1})$

N_0 is the density of droplets (for clouds, depending on the type and air pollution, from $\sim 10^8 - 10^9$ *droplets per cubic meter* (m^{-3}))
(it also gives us typical values of σ are then $\sim 10^{-2} - 10^{-1}\, m^{-1}$)

NOTE: When the light hits a particle we saw it could be absorbed or scattered.
The net effect is loss of energy (extinction), this is why σ in (**1**) is really the sum of 2 coefficients:

Extinction coefficient $\sigma_t = \sigma_a + \sigma_s$

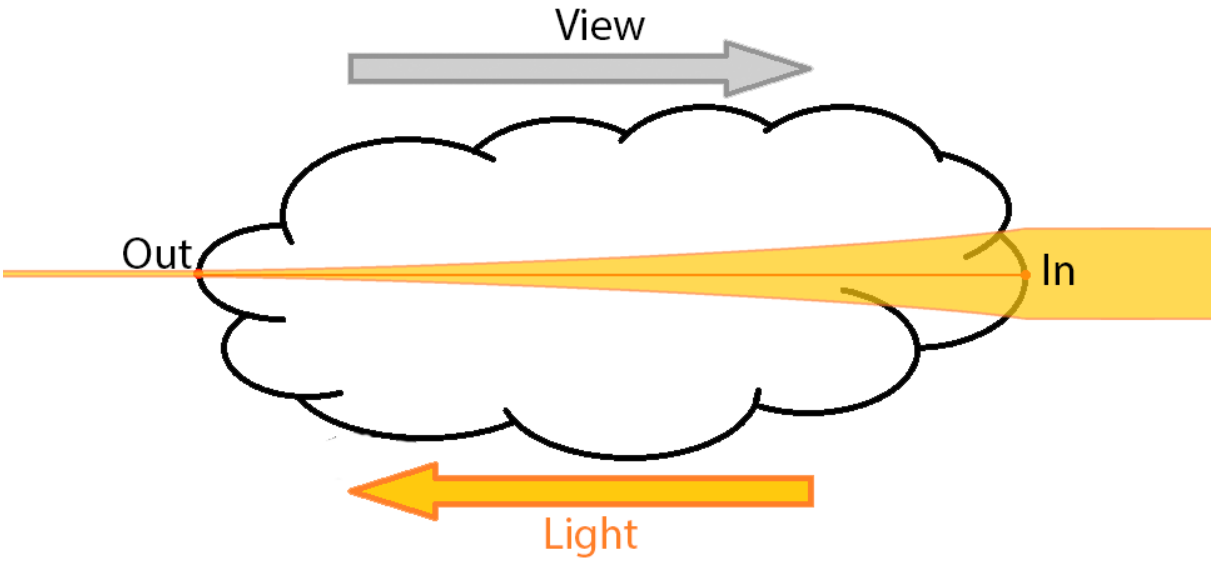
Where σ_a is the *Absorption coefficient* and σ_s is the *Scattering coefficient*.

Extinction

Now, here's one of the most important equations for rendering a participating medium: the *Extinction Function* also known as the Beer-Lambert law.

$$\tau(d) = e^{-\sigma_t d} \qquad (\mathbf{2})$$

This represents the probability to traverse the medium along a path of length **d** without hitting a particle. And incidentally, this also represents the transparency of the medium!



So, any radiance at distance Δ_x entering the cloud will see its intensity decrease:

$$L(\boldsymbol{x}, \boldsymbol{\omega}) = e^{-\sigma_t \Delta_x} L(\boldsymbol{x} + \Delta_x \boldsymbol{\omega}, \boldsymbol{\omega}) \qquad (3)$$

Here, $\boldsymbol{\omega}$ represents the view direction, $\boldsymbol{x}$ is the position where the light exits the volume (the "Out", in the figure) and Δ_x is the distance between In and Out (so $\boldsymbol{x} + \Delta_x \boldsymbol{\omega}$ is point "In" in the above figure).
Notice that $\boldsymbol{x}$ and $\boldsymbol{\omega}$ are written using a boldface font to mark the fact they are 3D vectors. Other quantities are scalars.

NOTE: Equation (**3**) can be considered valid as long as σ_t is supposed constant (i.e. as long as we agree a medium is homogeneous).

In-Scattering

So all right, we lose energy as we bump our way through molecules and particles. But there is also energy coming from somewhere else that may bump off in the direction we are viewing once again!

That would indeed add some energy and maybe even compensate the loss by extinction (imagine a magnifying glass bending light rays and concentrating energy onto a single spot).

Now, we need to estimate the chance for light to bounce from one particular direction off to another. Preferably, the direction we're looking at.

The Phase Function

It's almost the same as a BRDF for materials:

- ⇨ A black box indicating how light is interacting with the material
- ⇨ Difference with the BRDF is that it usually considers a single angle (the phase angle between 2 directions) and integrates to 1 over the entire sphere of directions.

$$\int_{\Omega_{4\pi}} p(\widehat{\boldsymbol{\omega_i}, \boldsymbol{\omega_o}})\, d\boldsymbol{\omega_o} = 1 \qquad \forall \boldsymbol{\omega_i}$$

Where $\theta = \widehat{\boldsymbol{\omega_i}, \boldsymbol{\omega_o}}$ is the angle between incoming and outgoing directions and $\Omega_{4\pi}$ represents the sphere of all possible directions.

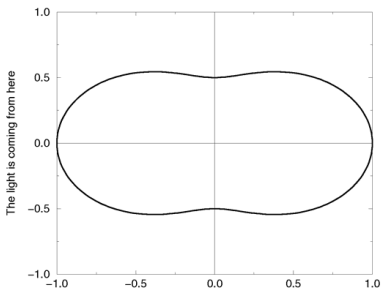
- ⇨ The "average cosine" of the phase function gives us g, the preferred scattering direction which will vary between -1 (backward scattering, light will be reflected back toward the source) to +1 (forward scattering, light will continue unaffected)

$$g = \int_{\Omega_{4\pi}} p(\theta) \cos(\theta)\, d\boldsymbol{\omega_o}$$

There exist some simple analytical models of phase functions.

Rayleigh Phase Function

When the considered particles are small enough, Rayleigh scattering is occurring. The phase function is peanut-shaped:



$$p(\theta) = \frac{3}{16\pi}(1 + \cos(\theta)^2)$$

The importance of scattering by molecules whose size is the same order of magnitude as the light's wavelength is proportional to the reciprocal of the wavelength's 4^{th} power:

$$I_{scattered} \propto \frac{1}{\lambda^4}$$

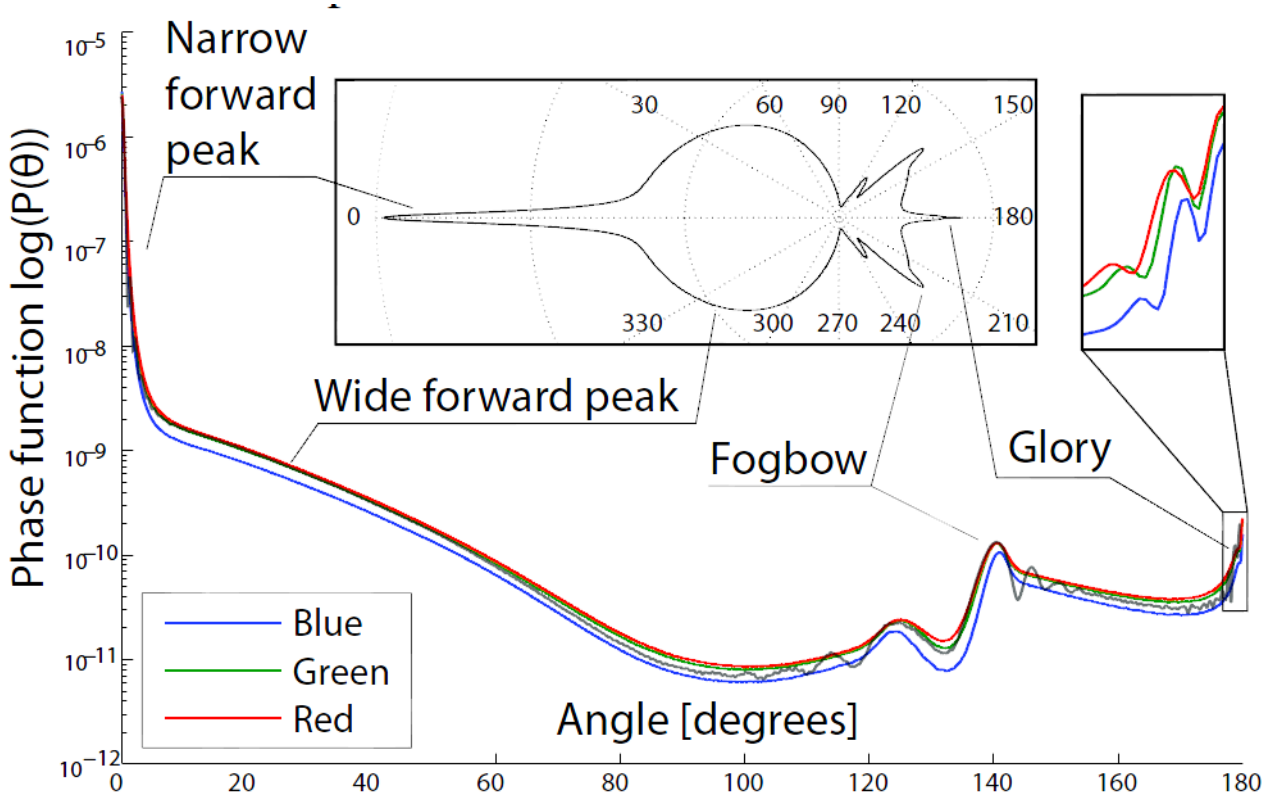
Meaning that scattering at $400nm$ is $\backsim 9$ times as great as that at $700nm$. This is the main reason why the sky is blue: shorter wavelengths (blue) are more scattered than long wavelengths (red).

Henyey-Greenstein Phase Function

For larger particles like pollutants, aerosols, dust and water droplets we must use Mie scattering.

Mie theory of scattering is very difficult to apprehend and the variety and complexity in shape and behaviors of the various components of the atmosphere usually make phase functions very difficult to work with.

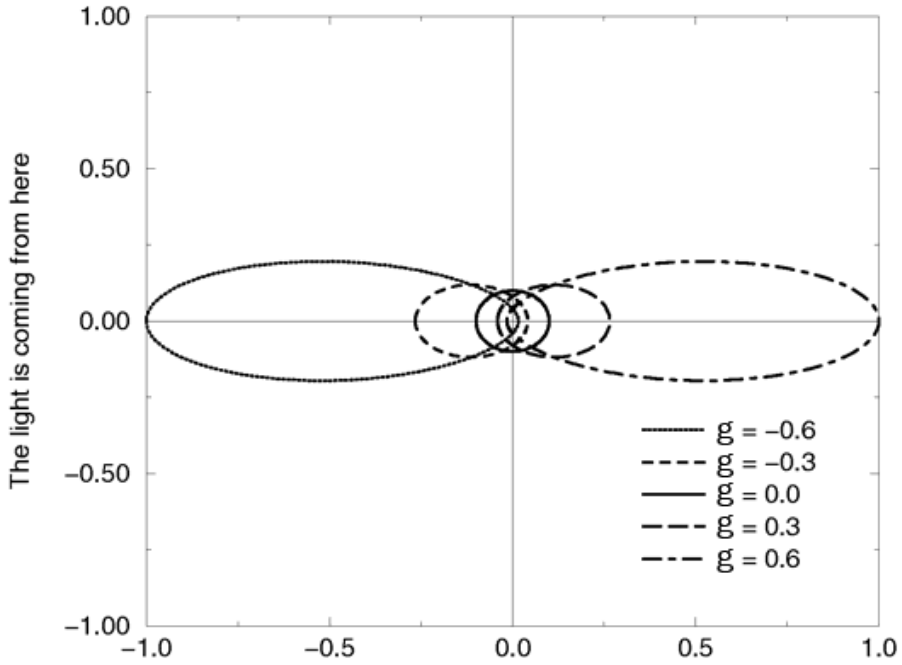
For example, here is what the average statistical phase function of a cloud would look like:



Instead of using that complex phase function, we usually choose a composition of multiple simpler functions called "Henyey-Greenstein" phase functions that look like this:

$$p_{HG}(\theta) = \frac{1 - g^2}{4\pi(1 + g^2 - 2gcos(\theta))^{3/2}}$$

We find g again, the preferred scattering direction that can vary in $[-1, +1]$.



NOTE: Because of the exponent $^3/_2$ the HG phase function is often replaced by the much cheaper Schlick phase function:

$$p_{Shlick}(\theta) = \frac{1 - k^2}{4\pi(1 + kcos(\theta))^2}$$

How to use them?

First, we must underline the fact that we're going to use phase functions only for single-scattering events: light that bounces only once off a particle. Multiple scattering events are very expensive and will be approximated another way as we will see later.

Phase functions can be cumulated using weights, assuming the sum of the weights is 1:

$$p(\theta) = w_0.p_{HG}(\theta) + w_1.p_{HG}(\theta) + (\dots) + w_i.p_{HG}(\theta) \quad \sum w_i = 1$$

So, in order to represent the complex Mie scattering from the figure above, we can simply use a weighted sum of simpler phase functions.

What about we start coding all this?

Armed with all the necessary knowledge, we must now focus on the code.

First of all, here are the equations we're dealing with:

$$L(x, \boldsymbol{\omega}) = \int_0^D e^{-\tau(x,x')} \sigma_s(x') \left[\int_{\Omega_{4\pi}} p(\boldsymbol{\omega}, \boldsymbol{\omega}') L_i(x', \boldsymbol{\omega}') \, d\boldsymbol{\omega}' \right] dx' \qquad (4)$$

$$\tau(x, x') = \int_x^{x'} \sigma_t(t) \, dt$$

$L(x, \boldsymbol{\omega})$ is the radiance at distance x (not a vector here!) along the direction $\boldsymbol{\omega}$ and D is the distance we need to trace inside the medium.

- The part in green is the extinction we saw earlier. It uses a new quantity, in brown, called the *Optical Depth* which represents the accumulated extinction coefficients along the path from x to x', multiplied by the length of that path.
  It simplifies nicely into $\tau(x) = \sigma_t(x) \, \Delta_x$ if σ_t is supposed constant along a path of length Δ_x but for this to work, we'll have to split the trace into small path (we'll come to that later).

- The part in blue is the in-scattering. We retrieve the phase function and a complex integral of light $L_i(x', \boldsymbol{\omega_i})$ coming from all possible directions (remember $\Omega_{4\pi}$ is the sphere of all possible directions).
  The tricky part here is this: $L_i(x', \boldsymbol{\omega_i})$ needs to be **itself** computed using equation **(4)**!
  It's a recursive relation, and each level of recursion accounts for a new order of scattering event.
  And this is the main reason why multiple scattering is very expensive and why we approximate it using a simple ambient term most of the time.
  (Note: this part is the equivalent of indirect lighting when performing global illumination)

In the figure above from [3], Bouthors et al. have taken into account scattering events up to order 20 (!!) and explain they are still relevant to the correct appearance of clouds!
This image is also the result of a compacting of a 25GB database computed in 7 days. So we can forget about it already! ☺

Let's Simplify

First of all, we split light sources into 3 categories:

- Direct Lighting from the Sun
- Indirect Lighting from the Sun
- Indirect Lighting from the Sky

Equation (4) then becomes:

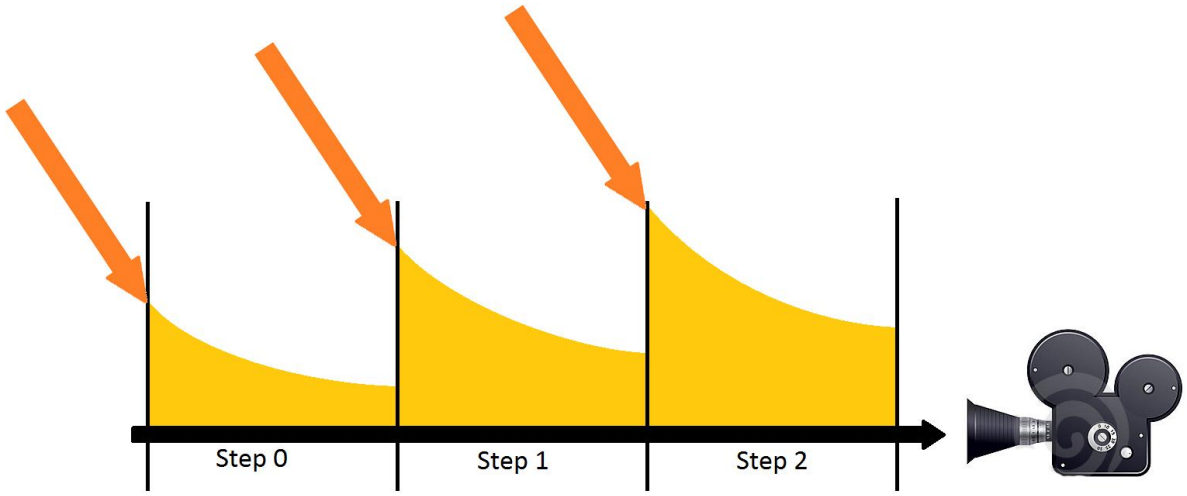
$$L(x,\boldsymbol{\omega}) = \int_0^D e^{-\tau(x,x')}\sigma_s(x')\ [p_{sun}(\boldsymbol{\omega},\boldsymbol{\omega_{sun}})L_{sun}(x',\boldsymbol{\omega_{sun}}) + p_{amb}L_{amb}]\ dx'$$

It's a little prettier without the inner integral. We brought back the complexity into our ability to determine a nice value for L_{amb} .

This integral can be read as:

1) Compute incoming light arriving at x' from the Sun and ambient environment
2) Scatter it using $\sigma_s(x')$

3) Let the viewer perceive only a small part of that light because of extinction between x and x'
4) Start again for the next small step dx'



We can easily discretize the integral into this new equation:

$$L(x,\boldsymbol{\omega}) \approx \sum_{i=1}^{N} e^{-\tau(x,x+i\,\Delta_x)}\,\sigma_s(x+i\,\Delta_x)\,[p_{sun}(\boldsymbol{\omega},\boldsymbol{\omega_{sun}})L_{sun}(x+i\,\Delta_x,\boldsymbol{\omega_{sun}}) + p_{amb}L_{amb}]\,\Delta_x$$

With $\Delta_x = \frac{D}{N}$ and N the amount of discrete steps we take inside the medium.

Now, to get rid of the annoying part in green we can notice that:

$$e^{-\tau(0,\Delta)} = e^{-\tau\left(0,\frac{\Delta}{2}\right)}.e^{-\tau\left(\frac{\Delta}{2},\Delta\right)}$$

Applying this recursively with our fixed steps Δ_x we get:

$$e^{-\tau(x,x+i\,\Delta_x)} \approx \prod_{j=1}^{i-1} e^{-\sigma_t(x+j\,\Delta_x)\,\Delta_x}$$

We see it becomes an accumulation of opacity in the form of a product of the opacities of each small slice. Easily something we could keep in a register on the side, and multiply with at each new step.
The final equations for our shader thus becomes:

$$L(x,\boldsymbol{\omega}) \approx \sum_{i=1}^{N} Ex_i\,\sigma_s(x+i\,\Delta_x)\,[p_{sun}(\boldsymbol{\omega},\boldsymbol{\omega_{sun}})L_{sun}(x+i\,\Delta_x,\boldsymbol{\omega_{sun}}) + p_{amb}L_{amb}]\,\Delta_x \quad (\mathbf{5})$$

$$Ex_i = Ex_{i-1}e^{-\sigma_t(x+i\,\Delta_x)\,\Delta_x}$$
$$Ex_0 = 1$$

The Shader Code

Here is the shader pseudo code for our ray marcher:

```
float3  Position = Start position at the beginning of the volume;
float3  View = Our normalized view direction;

float    Extinction = 1.0;   // We start with full transparency
float3  Scattering = 0.0;   // We start with no accumulated light

for each step
{
   float Density = SampleMediumDensity( Position );   // Sample a noise of some sort. Returns [0,1]
   float ScatteringCoeff = ScatteringFactor * Density;
   float ExtinctionCoeff = ExtinctionFactor * Density;

   // Accumulate extinction for that step
   Extinction *= exp( -ExtinctionCoeff * StepSize );

   // Compute in-scattered light
   float3  SunColor = ComputeSunColor( Position );  // Get Sun color arriving at position
   float3  AmbientColor = ComputeAmbientColor( Position, ExtinctionCoeff );  // Get ambient color at position
   float3  StepScattering = ScatteringCoeff * StepSize * (PhaseSun * SunColor + PhaseAmbient * AmbientColor);
   Scattering += Extinction * StepScattering;  // Accumulate scattering attenuated by extinction

   // March forward
   Position += StepSize * View;
}

return float4( Scattering, Extinction );
```

Notice that we return a float4 containing an alpha. But also notice that all along the ray-marching process we composed our scattered light color with a part of this alpha (the "Extinction * StepScattering" line).
We have a color pre-multiplied by its alpha, and thus the composition of that color with a background must use the pre-multiplied alpha blending operation [5]:

$$Dst' = Src + SrcAlpha * Dst$$

NOTE: In that example, we restricted ourselves by computing a monochromatic extinction stored as a single float but some events like Rayleigh scattering are actually wavelength dependent and in that case we must also consider the extinction as a float3 which will address the RGB components of the destination background (i.e. masking it with coloring).

We can no longer store a single Alpha value and thus need to use multiple render targets.

The Missing Parts

We're now left with some missing parts of the algorithm:

- ⇨ **SampleMediumDensity(pos)**, that computes the density of the medium at the given position
- ⇨ **ComputeSunColor(pos)**, that computes the lighting by the Sun at the given position
- ⇨ **ComputeAmbientColor(pos)**, that computes the ambient lighting at the given position

Computing the Density

Well this part is quite easy to implement as a Fractional Brownian Motion noise type that is well known to the demoscene (I've tried multiple types of noise (cellular, turbulence, ridged multifractal, etc.) and that's by far the best one for realistic clouds rendering).

```
static const float  AMPLITUDE_FACTOR = 0.707f;     // Decrease amplitude by √2/2 each new octave
static const float  FREQUENCY_FACTOR = 2.5789f;  // Increase frequency by some factor each new octave

float  SampleMediumDensity( float3 _Position )
{
  float3  UVW = _Position * 0.01;  // Let's start with some low frequency
  float   Amplitude = 1.0;
  float   V    = Amplitude * Noise( UVW );  Amplitude *= AMPLITUDE_FACTOR; UVW *= FREQUENCY_FACTOR;
          V += Amplitude * Noise( UVW );  Amplitude *= AMPLITUDE_FACTOR; UVW *= FREQUENCY_FACTOR;
          V += Amplitude * Noise( UVW );  Amplitude *= AMPLITUDE_FACTOR; UVW *= FREQUENCY_FACTOR;
          V += Amplitude * Noise( UVW );  Amplitude *= AMPLITUDE_FACTOR; UVW *= FREQUENCY_FACTOR;
          // Repeat as many times as necessary…

  return clamp( DensityFactor * V + DensityBias, 0, 1 );   // Factor and bias to help getting a nice result…
}
```

This part is essential and also very important to keep fast so I would advise to pack all the noise octaves in a large 3D texture but not too large either to avoid flushing your cache by too much tiling. To avoid obvious tiling the best I found is to use a low frequency 3D texture (something like 32^3) tiling veeery slowly to which you add detail with a larger 3D texture (something like 128^3) that tiles faster.

Computing the Sun's color

I left a big disappointment for the end! The algorithm is not that simple after all: we need some shadowing in our volume.
If we use a constant value for the light, not accounting for the sampling position within the volume, then the cloud will look quite unrealistic. Self-shadowing is of the utmost importance.

That means we need a shadow map, and one that stores volumetric shadowing at that!

Several techniques exist:

- ⇨ **Deep Shadow Maps [6]**
  - Not GPU friendly, we need to manage lists and perform sorting
  - Now possible with DX11 UAVs but not optimal
- ⇨ **Opacity Shadow Maps [7]**
  - Store Z where opacity reaches specified levels
  - Lacks precision
- ⇨ **Transmittance Function Maps [8]**

- Compact transmittance function using DCT basis
- Nice precision using only 6 coefficients + ZMin/ZMax stored into 2 render targets

In any case, we also need to ray-march the volume from the light's point of view.
The shader code that will create the shadow map should be the same as our ray-marcher before except that:

- ⇨ This time we ray-march from the point of view of the light, not the camera
- ⇨ We only account for extinction (or Transmittance) and don't care about scattering
- ⇨ We need to store/compact/encode that extinction at several key points in the volume, not a single value at the end

In the end, the pseudo-code using the volumetric shadow map is really simple:

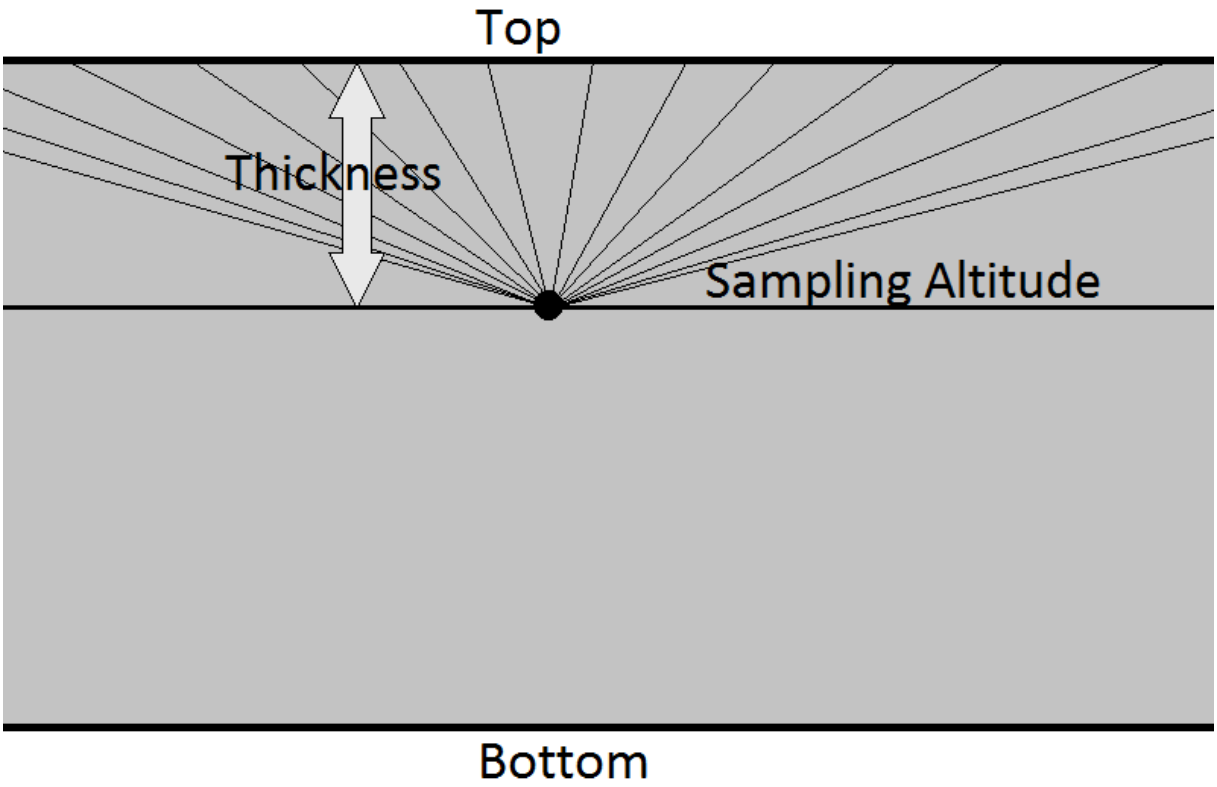
```
float3 ComputeSunColor ( float3 _Position )
{
  float3 ShadowMapPosition = _Position * World2ShadowMap;   // Transform from world to shadow map space
  float2 UV = ShadowMapPosition.xy;  // Our shadow map texture coordinates in [0,1]
  float  Z = ShadowMapPosition.z;    // Our depth in the shadow volume as seen from the light

  float  Extinction = GetShadowExtinction( UV, Z );   // Samples the shadow map and returns extinction in [0,1]

  return Extinction * SunColor;   // Attenuate sun color by extinction through the volume
}
```

Computing the Ambient color

The ambient color is very important to get right because it adds the realism we're lacking when considering only single scattering.

I'll show you a nice trick of mine that gives quite a nice result.

The idea is to think of the medium as an infinite slab of uniform density, the current density you sampled earlier in the ray-marching loop, then you assume you receive an ambient lighting from the sky (or the earth if you're considering the bottom slab) and the Sun that you "made ambient" by dividing its intensity by 4π (think of it as a isotropic phase function) and a factor that concentrates the loss due to high order scattering events.



We assume we have a homogeneous "ambient" radiance shining on our point P coming from the hemisphere of all possible directions that we will call L_{amb_+} for lighting from the top, and $L_{amb\_}$ for lighting from the bottom.

For isotropic lighting from top, we get :

$$L_+(\boldsymbol{x}) = \int_{\Omega_{2\pi+}} p_{iso}\, L_{amb_+}\, e^{-\sigma_t \frac{H_+}{\boldsymbol{n}_+ \cdot \boldsymbol{\omega}}}\, d\boldsymbol{\omega}$$

Where :

$L_+(\boldsymbol{x})$ is the top radiance

$\Omega_{2\pi+}$ is the top hemisphere of all possible directions

$p_{iso} = \frac{1}{4\pi}$ is the isotropic phase function (another assumption)

$\boldsymbol{n}_+$ is the local normal to the volume, pointing up

There is obviously a similar equation for the bottom part of the slab by replacing the + subscripts with the - sign.

Factoring out the constant terms and rewriting as a double integral over the hemisphere in spherical coordinates, we get:

$$L_+(\boldsymbol{x}) = p_{iso}\, L_{amb_+} \int_0^{2\pi} \int_0^{\frac{\pi}{2}} e^{-\sigma_t \frac{H_+}{\cos(\theta)}} \sin(\theta)\, d\theta\, d\varphi$$

$$L_+(\boldsymbol{x}) = p_{iso}\, L_{amb_+} 2\pi \int_0^{\frac{\pi}{2}} e^{-\frac{a}{\cos(\theta)}} \sin(\theta)\, d\theta$$

With $a = \sigma_t . H_+$

Fortunately, the integral has a closed form solution:

$$\int_0^{\frac{\pi}{2}} e^{-\frac{a}{\cos(\theta)}} \sin(\theta)\, d\theta = -\left[\cos(\theta) e^{-\frac{a}{\cos(\theta)}} + a\, Ei(-\frac{a}{\cos(\theta)})\right]_0^{\frac{\pi}{2}} = e^{-a} + a\, Ei(-a)$$

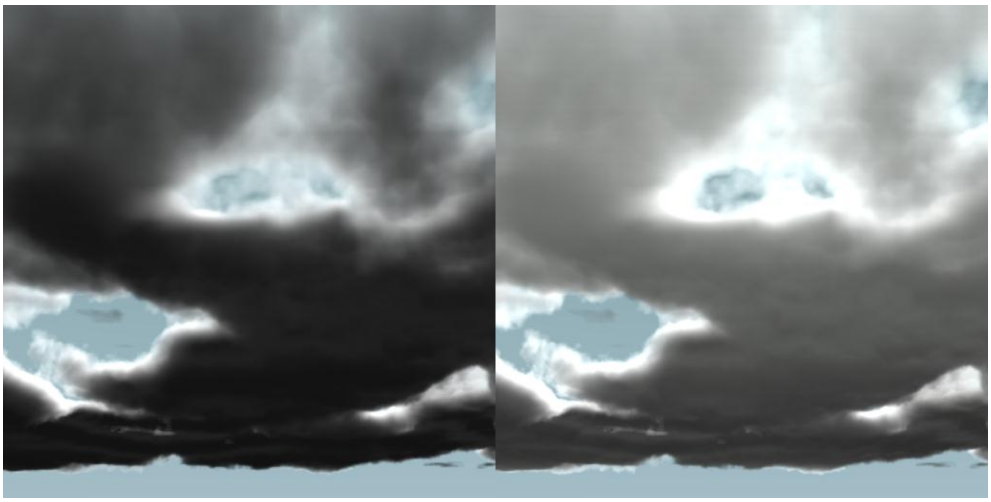
Where Ei is the exponential integral.

So here you go merrily with the pseudo-code:

```
// Exponential Integral
// (http://en.wikipedia.org/wiki/Exponential_integral)
float Ei( float z )
{
  return 0.577215664901532860606512 + log( 1e-4 + abs(z) ) + z * (1.0 + z * (0.25 + z * ( (1.0/18.0) + z * ( (1.0/96.0) + z *
(1.0/600.0) ) ) ) ); // For x!=0
}
float3  ComputeAmbientColor ( float3 _Position, float _ExtinctionCoeff )
{
   float  Hp = VolumeTop - _Position.y;   // Height to the top of the volume
   float  a = -_ExtinctionCoeff * Hp;
   float3  IsotropicScatteringTop = IsotropicLightTop * max( 0.0, exp( a ) - a * Ei( a ));

   float  Hb = _Position.y  - VolumeBottom;   // Height to the bottom of the volume
          a = -_ExtinctionCoeff * Hb;
   float3  IsotropicScatteringBottom = IsotropicLightBottom * max( 0.0, exp( a ) - a * Ei( a ));


   return IsotropicScatteringTop  + IsotropicScatteringBottom;
}
```

Values for IsotropicLightTop are coming from the Sky and “Sun made ambient”, values for IsotropicLightBottom can also add the contribution of Sun light that bounced off the ground so you can simulate color bleeding.

We can see the improvement here:



Bonus: Emissive Medium

On top of absorption and scattering events, you could also assume your medium is emitting some light (explosion, flame, etc.).

This is really easy to add to your ray-marching loop if you simply add an *emissive term* to the StepScattering variable.
This term could be linked to the temperature of the medium (black body radiation) or depend on a light source lying in the medium.

Conclusion

As a summary, the steps you need to take to render a participating medium are:

1) Render a shadow map with volumetric information about extinction
   ➔ This requires to ray-march the volume from the light's point of view
2) Render the volume by ray-marching
   ➔ Accumulate lighting by querying the volumetric shadow map
   ➔ Perform extinction along the ray
3) Compose the resulting image (Scattering + Extinction) with the background using the pre-multiplied alpha blend mode

Typically, a minimum of 16 steps of ray-marching are necessary to render the shadow map while 64 or more steps may be needed to render the volume itself, depending on the level of detail you wish to attain.

Nowadays, one can easily attain several hundreds of frames per second for a fullscreen, well detailed rendering of an entire sky as it is the case with Nuaj', my atmosphere renderer for Unity.

The physical processes behind extinction and scattering can also be employed to render many other mediums like water, skin or translucent objects.

Hopefully, with a little practice, you will be able to master these new tools and make volumetric rendering a common practice in the demoscene or game industry.

References:

[1] *"Physics and Math of Shading"*, Naty Hoffman. Siggraph 2012

[2] *"Real-time realistic illumination and shading of stratiform clouds"*, Bouthors et al., Eurographics Workshop on Natural Phenomena - 2006

[3] *"Interactive multiple anisotropic scattering in clouds"*, Bouthors et al., ACM Symposium on Interactive 3D Graphics and Games (I3D) – 2008

[4] "Cloud liquid water content, drop sizes, and number of droplets"

[5] "Pre-Multiplied Alpha", Tom Forsyth - 2006

[6] "Deep Shadow Maps", Lokovic & Veach - 2000

[7] "Opacity Shadow Maps", Kim & Neumann – 2001

[8] "Transmittance Function Mapping", Delalandre et al., I3D '11 Symposium on Interactive 3D Graphics and Games – 2011