II

4

Real-Time Volumetric Cloudscapes

Andrew Schneider

4.1 Overview

Real-time volumetric clouds in games usually pay for fast performance with a reduction in quality. The most successful approaches are limited to low-altitude fluffy and translucent stratus-type clouds. We propose a volumetric solution that can fill a sky with evolving and realistic results that depict high-altitude cirrus clouds and all of the major low-level cloud types, including thick, billowy cumu-

Figure 4.1. Several cloudscapes that were drawn in real time for the game *Horizon: Zero Dawn*.

lus clouds. Additionally, our approach approximates several volumetric lighting effects that have not yet been present in real-time cloud rendering solutions. And finally, this solution performs well enough in memory and on the GPU to be included in a AAA console game. (See Figure 4.1.)

4.2 Introduction

The standard solutions for rendering clouds in AAA console games involve assets of some kind, either 2D billboards, polar sky dome images, or volumetric libraries that are instanced at render time. For games that require a constantly changing sky and allow the player to cover vast distances, such as open world, the benefits of highly detailed assets are overshadowed by the cost of storing and accessing data for multiple camera angles, times of day, and lighting conditions. Additionally, the simulation of cloud system evolution is limited to tricks or fakes such as rotating the sky dome or distorting images using 2D noise.

Numerous techniques for procedural cloud systems do not rely on assets. Several good examples are freely available on ShaderToy.com, such as "Clouds" [Quilez 13]. Evolution studios used middleware called TrueSky to deliver impressive atmospheric weather effects for the game *Drive Club* [Simul 13].

Yet, there are several limitations with these approaches:

- They all only describe low-altitude stratus clouds and not the puffy and billowy stratocumulus or cumulus clouds.
- Current volumetric methods do not implement realistic lighting effects that are specific to clouds.
- Real-time volumetric clouds are often quite expensive in terms of performance and memory and are not really worth the quality of the results produced.

For the game *Horizon: Zero Dawn*, we have developed a new solution that addresses these problems. We submit new algorithms for modeling, lighting, and rendering, which deliver realistic and evolving results while staying within a memory budget of 20 MB and a performance target of 2 ms.

4.3 Cloud Modeling

Figure 4.2 shows the various cloud types and their height ranges. There are two layers that we render volumetrically: the low stratus clouds, which exist between 1.5 km and 4 km, and the cumulonimbus clouds, which span the entire lower atmosphere from 1 km to 8 km. The alto and cirro class clouds are usually very thin in height and can be rendered for less expense with a 2D texture lookup.

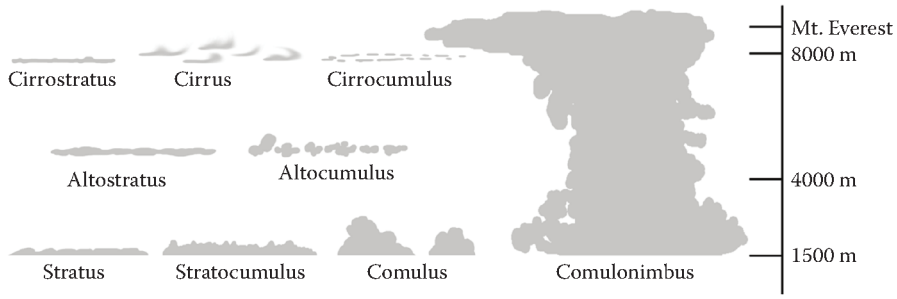


Figure 4.2. A map of the major cloud types.

As the day progresses and the sun heats the earth, water vapor rises from the surface and travels through these layers of atmosphere. Each layer has its own wind direction and temperature. As the vapor travels higher in the atmosphere, the temperature decreases. As temperature decreases the vapor condenses into water or ice around particles of dust it encounters. (Sometimes this comes back down as rain or snow.) The great deal of instability in the flow of this vapor introduces turbulence. As clouds rise, they tend to make billowing shapes. As they diffuse, they stretch and dissipate like fog [Clausse and Facy 61].

Clouds are really amazing examples of fluid dynamics in action, and modeling this behavior requires that the designer approach clouds in a way that approximates the underlying physics involved. With these concepts in mind, we define several techniques that will be used in our ray march to model clouds.

Sections 4.3.1 through 4.3.3 detail some concepts that are used to model clouds and Section 4.3.4 explains how they are all used together.

4.3.1 Modified Fractal Brownian Motion

The standard approach for modeling volumetric cloud systems in real time involves using a ray march with a technique called *fractal Brownian motion*, or FBM for short [Mandelbrot and van Ness 68]. (See Figure 4.3.) FBM is the sum of a series of octaves of noise, each with higher frequency and lower amplitude.

Perlin noise [Perlin 85] is commonly used for this purpose. While this is a reliable model for producing the fog-like shapes of stratus clouds, it fails to describe the round, billowy shapes in cumulus clouds or give them an implied sense of motion as seen in Figure 4.4.

Perlin noise can be flipped over in the middle of its range to create some puffy shapes, but because it is just one flavor of noise, it still lacks the packed cauliflower pattern seen in clouds. Figure 4.5 shows Perlin noise, the result of `abs(Perlin * 2 + 1)`, and photographic reference of the fractal billowing pattern found in clouds.

Figure 4.3. Procedural clouds generated with a ray march and an FBM noise.

Figure 4.4. Photographic reference showing round billowing shapes, similar to puffs of smoke from a factory vent.

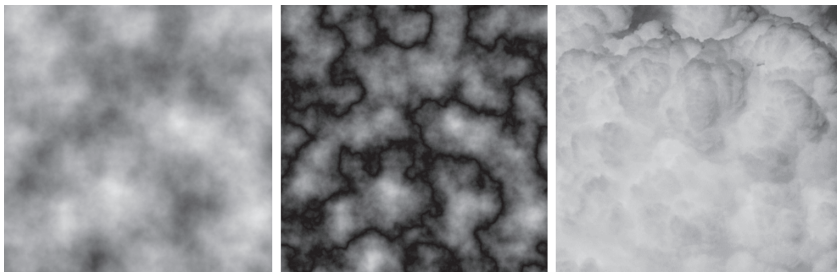


Figure 4.5. Seven-octave Perlin noise (left), Perlin noise made to look "puffy" (center), and photographic reference of the packed cauliflower shapes in clouds (right).

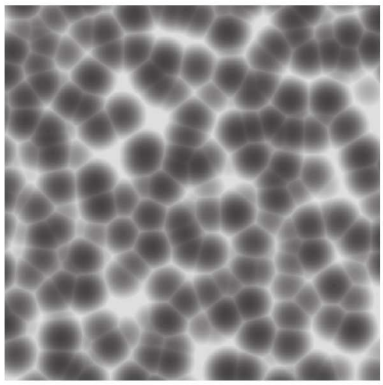


Figure 4.6. Worley noise.

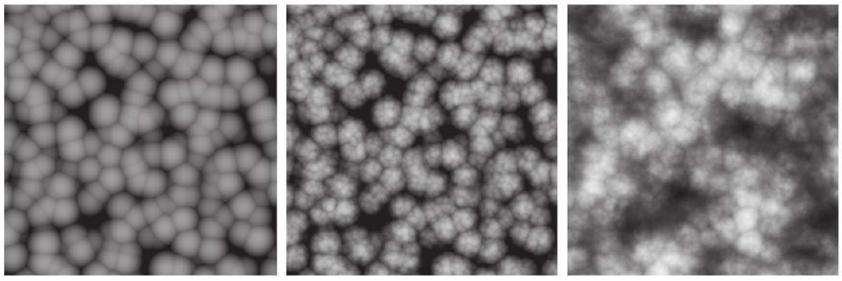


Figure 4.7. Inverted Worley noise (left), FBM composed of Worley noise (center), and Perlin-Worley noise (right).

Another flavor of noise, Worley noise, was introduced in 1996 by Steven Worley [Worley 96] and is often used in rendering caustics and water effects, as seen in Figure 4.6.

If inverted and used in a FBM, Worley noise approximates a nice fractal billowing pattern. It can also be used to add detail to the low-density regions of the low-frequency Perlin noise. (See Figure 4.7, left and center.) We do this by remapping the Perlin noise using the Worley noise FBM as the minimum value from the original range.

```
OldMin = Worley_FBM
PerlinWorley = NewMin + (((Perlin - OldMin) / (OldMax - OldMin))
     * (NewMax - NewMin))
```

This method of combining the two noise types adds a bit of billowing to the connectedness produced in Perlin noise and produces a much more natural result.

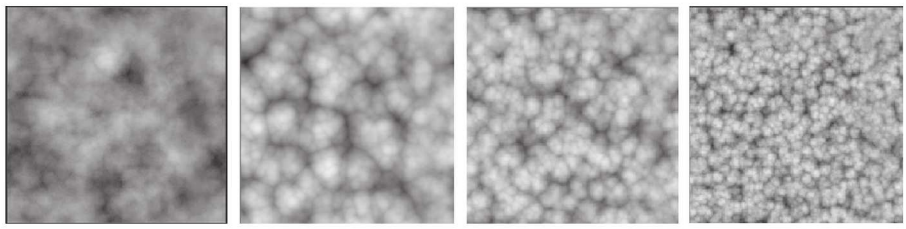


Figure 4.8. A slice of the low-frequency noise's RGBA channels. The first slice is Perin-Worley noise. The last three are Worley noises at increasing frequencies. (Resolution: 128^3.)

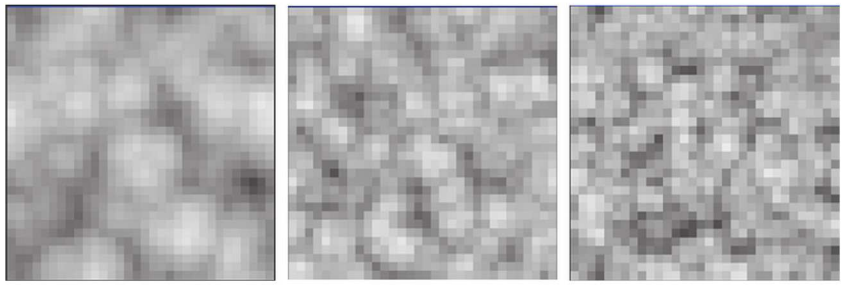


Figure 4.9. From left to right, a slice of the high-frequency noise's RGB channels and Worley noise at increasing frequencies. (Resolution: 32^3.)

We refer to this as our low frequency Perlin-Worley noise and it is the basis for our modeling approach. (See Figure 4.7, right.)

Instead of building the FBM using one texture read per octave, we precompile the FBM so we only have to read two textures. Figure 4.8 shows our first 3D texture, which is made of the Perlin-Worley noise FBM and three octaves of Worley noise FBM's. Figure 4.9 shows our second 3D texture, which consists of three more octaves of Worley noise.

The first 3D texture defines our base cloud shape. The second is of higher frequency and is used to erode the edges of the base cloud shape and add detail, as explained further in Section 4.3.4.

4.3.2 Density-Height Functions

Previous work in this area creates a specific cloud type by biasing or scaling the cloud density value, based on height [Quilez 13].

This function is used to bias or scale the noise signal and produce a cloud. This has limited the types of clouds seen in other work to one type because the maximum height of the clouds never changes.

Figure 4.10. The gradients produced by three density height functions to represent stratus (left), cumulus (center), and cumulonimbus (right) clouds.

Figure 4.11. Results of three functions used to represent stratus (left), cumulus (center), and cumulonimbus (right) clouds.

We extend this approach by using three such functions, one for each of the three major low-level cloud types: stratus, stratocumulus, and cumulus. Figure 4.10 shows the gradient functions we used. Figure 4.11 shows the results of using these functions to change cloud density over height.

At runtime we compute a weighted sum of the three functions. We vary the weighting using a weather texture to add more or less of each cloud type—details are in the next section.

4.3.3 Weather Texture

For our purposes we want to know three things at any point in the domain of our cloud system:

1. Cloud coverage: The percentage of cloud coverage in the sky.
2. Precipitation: The chance that the clouds overhead will produce rain.
3. Cloud type: A value of 0.0 indicates stratus, 0.5 indicates stratocumulus, and 1.0 indicates cumulus clouds.

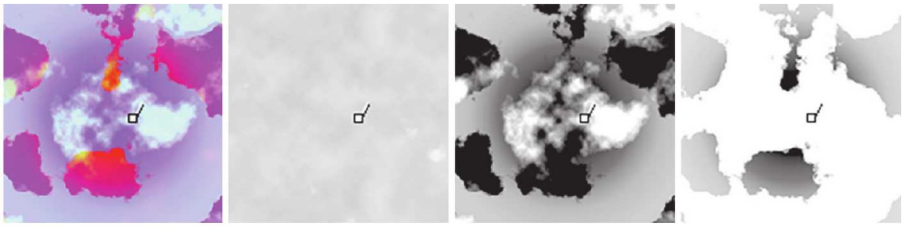


Figure 4.12. Weather texture (left), then (from left to right) coverage signal (red), precipitation signal (green), and cloud type signal (blue).

These attributes can all be expressed as a probability between zero and one, which makes them easy to work with and to preview in a 2D texture. This buffer can be sampled to get a value for each attribute at any point in world space.

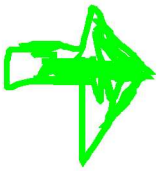
Figure 4.12 breaks the weather map for this scene down into its components. The scale of this map is 60,000 × 60,000 meters, and the arrows indicate camera direction.

In reality, rain clouds are always present where it is raining. To model this behavior, we bias cloud type to cumulonimbus and cloud coverage to at least 70% where the chance of rain is 100%.

Additionally, we allow the artist to override the weather texture to produce art-directed skies for cutscenes or other directed experiences [Schneider 15, slide 47].

4.3.4 Cloud Sampler

Having established the components of the cloud density function, we will now move on to the cloud model.



Like all other volumetric cloud solutions to date, we use a ray march. A ray march takes steps through a domain and samples density values for use in lighting and density calculations. These data are used to build the final image of the volumetric subject. Our cloud density sample function does most of the work of interpreting the sample position and the weather data to give us the density value of a cloud at a given point.

Before we start working in the function, we calculate a normalized scalar value that represents the height of the current sample position in the cloud layer. This will be used in the last part of the modeling process.

```
// Fractional value for sample position in the cloud layer.
float GetHeightFractionForPoint(float3 inPosition,
                                float2 inCloudMinMax)
{
    // Get global fractional position in cloud zone.
    float height_fraction = (inPosition.z - inCloudMinMax.x ) /
```

```
                                (inCloudMinMax.y - inCloudMinMax.x);

    return saturate(height_fraction);
}
```

We also define a remapping function to map values from one range to another, to be used when combining noises to make our clouds.

```
// Utility function that maps a value from one range to another.
float Remap(float original_value, float original_min,
            float original_max, float new_min, float new_max)
{
    return new_min + (((original_value - original_min) /
            (original_max - original_min)) * (new_max - new_min))
}
```

The first step of our sampling algorithm is to build a basic cloud shape out of the low-frequency Perlin-Worley noise in our first 3D texture. The process is as follows:

1. The first step is to retrieve the four low-frequency noise values required to build a basic cloud shape. We sample the first 3D texture, containing low-frequency octaves.

2. We will use the first channel, which contains the Perlin-Worley noise, to establish our base cloud shape.

3. Though the basic Perlin-Worley noise provides a reasonable cloud density function, it lacks the detail of a realistic cloud. We use a remapping function to add the three other low-frequency noises to the edges of the Perlin-Worley noise. This method of combining noises prevents the interior of the Perlin-Worley cloud shape from becoming non-homogenous and also ensures that we only add detail in the areas that we can see.

4. To determine the type of cloud we are drawing, we compute our density height function based on the cloud type attribute from our weather texture.

5. Next, we multiply the base cloud shape by the density height function to create the correct type of cloud according to the weather data.

Here is how it looks in code:

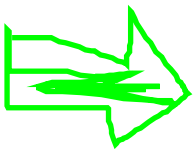
```
float SampleCloudDensity(float3 p, float3 weather_data)
{
    // Read the low-frequency Perlin-Worley and Worley noises.
    float4 low_frequency_noises = tex3Dlod(Cloud3DNoiseTextureA,
        Cloud3DNoiseSamplerA, float4 (p, mip_level) ).rgba;

    // Build an FBM out of  the low frequency Worley noises
```

Figure 4.13. The low-frequency "base" cloud shape.

```
// that can be used to add detail to the low-frequency
// Perlin-Worley noise.
float low_freq_FBM = ( low_frequency_noises.g * 0.625 )
                   + ( low_frequency_noises.b * 0.25 )
                   + ( low_frequency_noises.a * 0.125 );

// define the base cloud shape by dilating it with the
// low-frequency FBM made of Worley noise.
float base_cloud = Remap( low_frequency_noises.r, -
     ( 1.0 -  low_freq_FBM), 1.0, 0.0, 1.0 );

// Get the density-height gradient using the density height
// function explained in Section 4.3.2.
float density_height_gradient =
     GetDensityHeightGradientForPoint( p, weather_data );

// Apply the height function to the base cloud shape.
base_cloud *=  density_height_gradient;
```

At this point we have something that already resembles a cloud, albeit a low-detail one (Figure 4.13).

Next, we apply the cloud coverage attribute from the weather texture to ensure that we can control how much the clouds cover the sky. This step involves two operations:



1. To make the clouds realistically grow as we animate the coverage attribute, we expand the base cloud shape that was produced by the previous steps using the cloud coverage attribute in the remapping function.
2. To ensure that density increases with coverage in an aesthetically pleasing way, we multiply this result by the cloud coverage attribute.

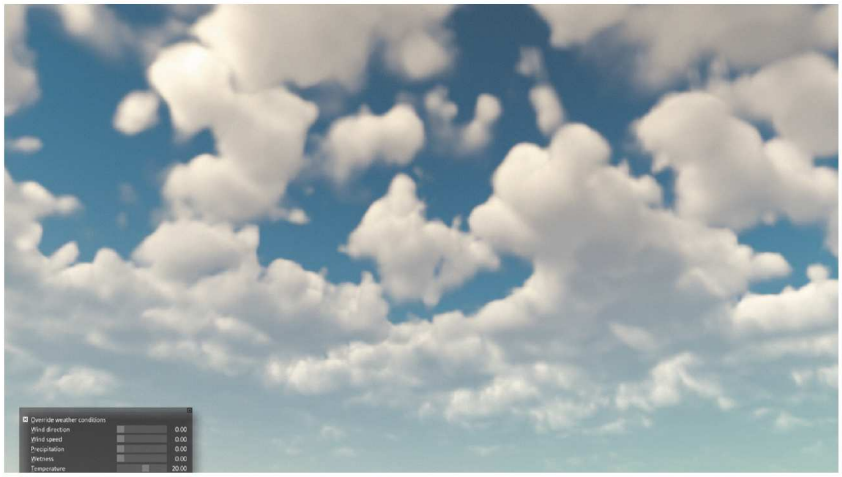


Figure 4.14. The "base" cloud shape with coverage applied.

Here is how it looks in code:

```
// Cloud coverage is stored in weather_data's red channel.
float cloud_coverage = weather_data.r;

// Use remap to apply the cloud coverage attribute.
float base_cloud_with_coverage   = Remap(base_cloud,
    cloud_coverage, 1.0, 0.0, 1.0);
// Multiply the result by the cloud coverage attribute so
// that smaller clouds are lighter and more aesthetically
// pleasing.
base_cloud_with_coverage *= cloud_coverage;
```

The result of these steps is shown in Figure 4.14. The base cloud is still low detail but it is beginning to look more like a system than a field of noise.

Next, we finish off the cloud by adding realistic detail ranging from small billows created by instabilities in the rising water vapor to wispy distortions caused by atmospheric turbulence (see examples in Figure 4.15).

We model these effects using three steps:

1. We use animated curl noise to distort the sample coordinate at the bottom of the clouds, simulating the effect of turbulence when we sample the high-frequency 3D texture using the distorted sample coordinates.

2. We build an FBM out of the high-frequency Worley noises in order to add detail to the edges of the cloud.

3. We contract the base cloud shape using the high-frequency FBM. At the base of the cloud, we invert the Worley noise to produce wispy shapes in

Figure 4.15. Photographic reference of billowy shapes and wispy shapes created by atmospheric turbulence.

this region. Contracting with Worley noise at the top produces billowing detail.

Here is how it looks in code:

```
// Add some turbulence to bottoms of clouds.
p.xy += curl_noise.xy * (1.0 - height_fraction);

// Sample high-frequency noises.
float3 high_frequency_noises = tex3Dlod(Cloud3DNoiseTextureB,
    Cloud3DNoiseSamplerB,  float4 (p * 0.1, mip_level) ).rgb;

// Build-high frequency Worley noise FBM.
float high_freq_FBM = ( high_frequency_noises.r * 0.625 )
                    + ( high_frequency_noises.g * 0.25 )
                    + ( high_frequency_noises.b * 0.125 );

// Get the height_fraction for use with blending noise types
// over height.
float height_fraction  = GetHeightFractionForPoint(p,
                                                   inCloudMinMax);

// Transition from wispy shapes to billowy shapes over height.
float high_freq_noise_modifier = mix(high_freq_FBM,
       1.0 - high_freq_FBM, saturate(height_fraction * 10.0));

// Erode the base cloud shape with the distorted
```

Figure 4.16. The final cloud shape.

```
    // high-frequency Worley noises.
    float final_cloud = Remap(base_cloud_with_coverage,
        high_freq_noise_modifier * 0.2 , 1.0, 0.0, 1.0);

    return final_cloud;
}
```

The result of these steps is shown in Figure 4.16. This series of operations is the framework that our sampler uses to create cloudscapes in the ray march, but we take additional steps to add that implied sense of motion that traditional noise-based solutions for cloudscapes lack.

To simulate the shearing effect as a cloud rises from one atmosphere layer to another, we offset the sample position in the wind direction over altitude. Additionally, both 3D texture samples are offset in the wind direction and slightly upward over time, but at different speeds. Giving each noise its own speed produces a more realistic look to the motion of the clouds. In a time lapse, the clouds appear to grow upward.

```
    // Wind settings.
    float3 wind_direction = float3(1.0, 0.0, 0.0);
    float cloud_speed = 10.0;

    // cloud_top offset pushes the tops of the clouds along
    // this wind direction by this many units.
    float cloud_top_offset = 500.0;

    // Skew in wind direction.
    p += height_fraction * wind_direction * cloud_top_offset;
```

Figure 4.17. Sample cloudscapes, captured on the Playstation 4.

```
// Animate clouds in wind direction and add a small upward
// bias to the wind direction.
p += (wind_direction + float3(0.0, 0.1, 0.0)  ) * time
     * cloud_speed;
```

This code must be located before any 3D texture samples in the `CloudDensitySample()` function.

4.3.5 Results

Some volumetric cloudscapes created using different weather settings are illustrated in Figure 4.17.

This modeling approach allows us to sculpt numerous unique cloudscapes. When a rain signal approaches the camera along the wind direction, it gives the effect of an approaching storm front [Schneider 15, slide 43–44].

4.4 Cloud Lighting

Volumetric cloud lighting is a very well researched area of computer graphics. Unfortunately for game developers, the best results come from taking high numbers of samples. This means that we have to find ways to approximate the complicated and expensive processes that take place when producing film-quality clouds.

Figure 4.18. Photographic reference of directional scattering (left), the silver lining effect (center), and the dark edges effect (right).

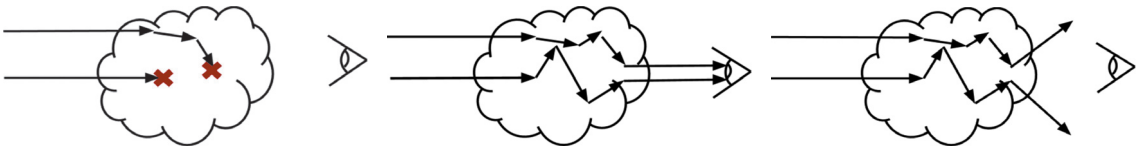


Figure 4.19. Examples of three light behaviors in a cloud: absorption (left), in-scattering (center), and out-scattering (right).

There are three effects in particular for which our approach solves with approximations: the multiple scattering and directional lighting in clouds, the silver lining effect when we look toward the sun, and the dark edges on clouds when we look away from the sun. Figure 4.18 shows photographic references of these three effects.

4.4.1 Volumetric Scattering

When light enters a cloud, the majority of the light rays spend their time refracting through water droplets and ice inside of the cloud before scattering toward our eyes [Van De Hulst 57]. There are three things that can happen to a photon entering a cloud (see also Figure 4.19):

1. It can be absorbed by water or non-participating particles in the cloud such as dust; this is *extinction* or *absorption*.
2. It can exit the cloud toward the eye; this is *in-scattering*.
3. It could exit the cloud traveling away from the eye; this is *out-scattering*.

Beer's law is a standard method for approximating the probability of each of these three outcomes.

4.4.2 Beer's Law

Originally conceived of as a tool for chemical analysis, Beer's law models the attenuation of light as it passes through a material [Beer 52]. (See Figure 4.20.)

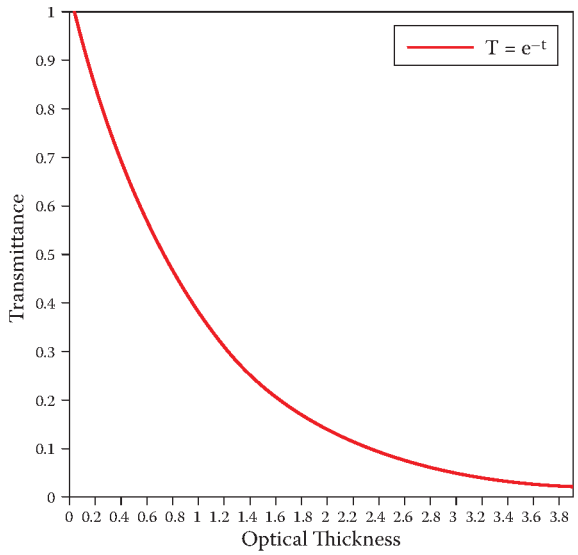


Figure 4.20. Beer's law: Transmittance as a function of optical depth.

In the case of volumetrics, it can be used to reliably calculate transmittance based on optical thickness [Wrenninge 13].

If our participating media are non-homogenous, like clouds, we must accumulate optical thickness along the light ray using a ray march. This model has been used extensively in film visual effects, and it forms the foundation of our lighting model.

Here is how it is implemented in code:

```
light_energy = exp( - density_samples_along_light_ray );
```

4.4.3 Henyey-Greenstein Phase Function

In clouds, there is a higher probability of light scattering forward [Pharr and Humphreys 10]. This is responsible for the silver lining in clouds. (See Figure 4.21.)

In 1941, the Henyey-Greenstein phase function was developed to mimic the angular dependence of light scattering by small particles, which was used to describe scattering of light by interstellar dust clouds [Henyey and Greenstein 41]. In volumetric rendering the function is used to model the probability of light scattering within participating media. We use a single Henyey-Greenstein phase function with an eccentricity (directional component) g of 0.2, to make sure that

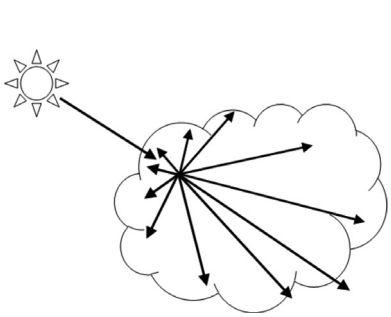


Figure 4.21. Illustration of forward scattering of light in a cloud (left), and photographic reference of the silver lining effect (right).

HenyeyGreenstein, modela la probabilidad de baja profundidad en bordes. Beer, modela el atenuamiento de la luz en base a la profundidad.
Más no la dispersion de la luz.

more light in our clouds scatters forward:

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

And here is how it looks implemented in code:

```
float HenyeyGreenstein(float3 inLightVector, float3 inViewVector,
                       float inG)
{
    float cos_angle = dot(normalize(inLightVector),
                          normalize(inViewVector));
    return ((1.0 - inG * inG) / pow((1.0 + inG * inG -
         2.0 * inG * cos_angle), 3.0 / 2.0)) / 4.0 * 3.1415;
}
```

The results are shown in Figure 4.22.

Figure 4.22. Clouds without the Henyey-Greenstein phase function (left), and clouds with the Henyey-Greenstein phase function (right).

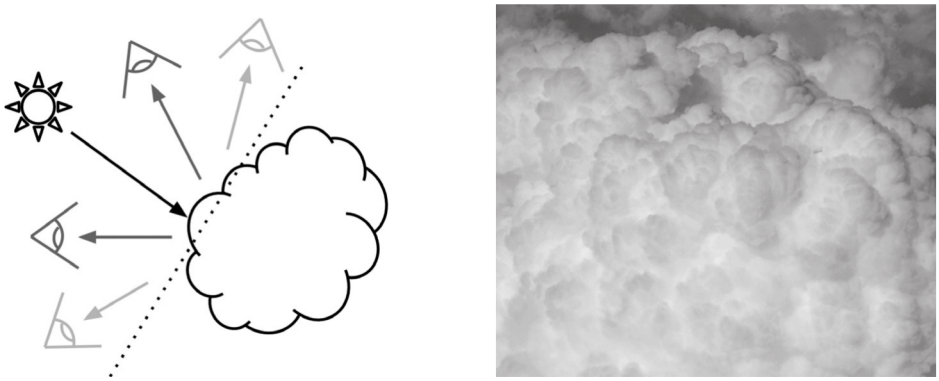


Figure 4.23. A diagram showing the 180-degree view angle where the dark edge effect is apparent (left), and photographic reference of the dark edge effect (right).

4.4.4 In-Scattering Probability Function (Powdered Sugar Effect)

Beer's law is an extinction model, meaning that it is concerned with how light energy attenuates over depth. This fails to approximate an important lighting effect related to in-scattering on the sun-facing sides of clouds. This effect presents itself as dark edges on clouds when a view ray approaches the direction of the light ray. There is a similar effect in piles of powdered sugar, the source of our nickname for this effect. See Figure 4.23 for an illustration.

This effect is most apparent in round, dense regions of clouds, so much so that the creases between each bulge appear brighter than the bulge itself, which is closer to the sun. These results would appear to be the exact opposite of what Beer's law models.

Recall that in-scattering is an effect in which light rays inside a cloud bounce around until they become parallel and then exit the cloud and travel to our eyes. This phenomenon even occurs when we look at a sunlit side of a cloud (Figure 4.24).

Also recall that more light scatters forward, along the original light ray direction, due to forward scattering. However, a relatively large optical depth must exist for there to be a reasonable chance for a photon to turn 180 degrees. Paths around the edge of the cloud won't pass through a sufficiently large optical depth to turn a noticeable fraction of the photons completely around. Paths that do have an optical depth large enough to turn a photon 180 degrees are almost always well inside the cloud, so Beer's law extinction will kill this contribution before it leaves the cloud toward our eye. Crevices and cracks are an exception; they provide a window into the interior of the cloud volume where there are photon paths with relatively large optical depths, allowing a low-density shortcut for photons to escape, making the crevices brighter than the surrounding bulges.

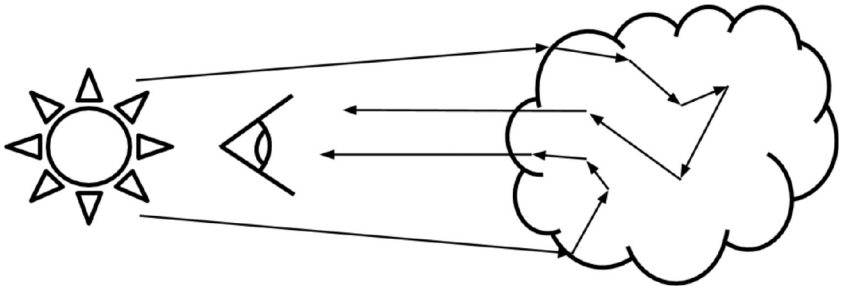


Figure 4.24. An illustration of in-scattering producing a 180-degree turn in the incoming light rays.

We chose to express this phenomenon as a probability. Imagine you are looking at one of these bulgy regions on a cloud at the same angle as a group of light rays coming from the sun behind you (Figure 4.25).

If we sample a point just below the surface on one of the bulges and compare it to a point at the same depth in one of the crevices, the point in the crevice will have more potential cloud material that can contribute to in-scattering (Figure 4.26). In terms of probability, the crease should be brighter.

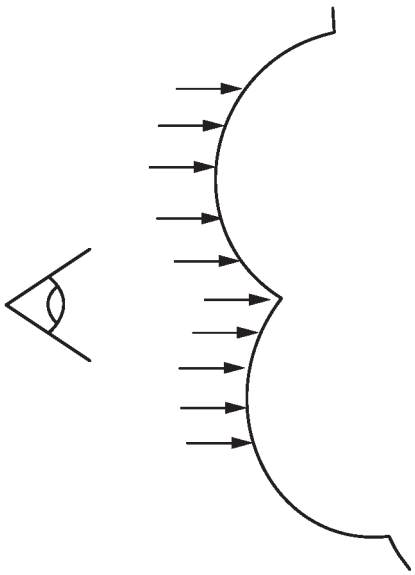


Figure 4.25. Light hitting bulges on a cloud.

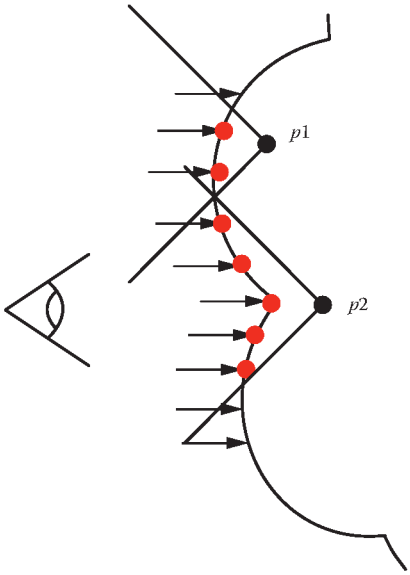


Figure 4.26. Illustration showing higher in-scatter potential for the creases.

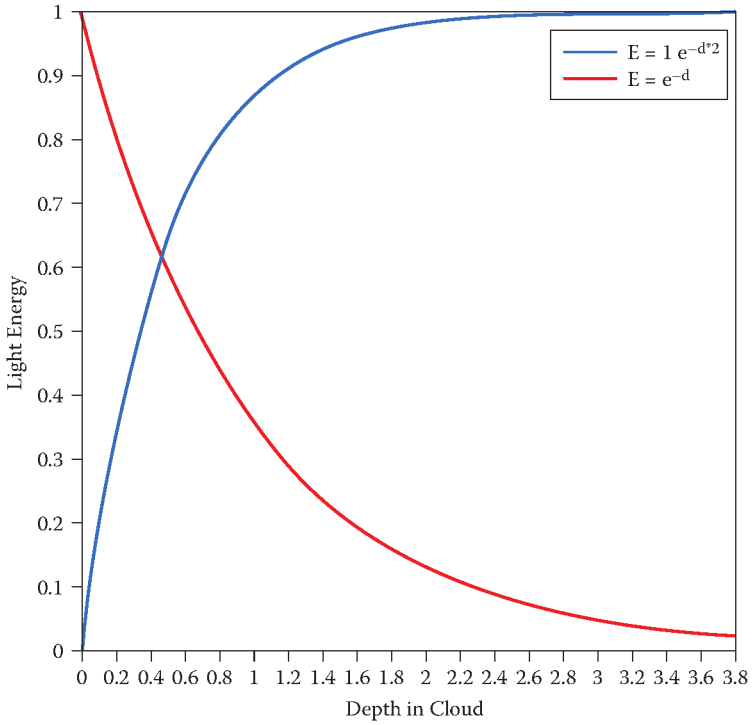


Figure 4.27. Beer's law compared to our approximation for the powdered sugar effect.

Using this thought experiment as a guide, we propose a new function to account for this effect. Because this result is effectively the opposite of Beer's law, we represent it as an inverse of the original function (Figure 4.27).

For our purposes this is an accurate enough approximation of this phenomenon, which does not require any additional sampling.

We combine the two functions into a new function: Beer's-Powder. Note that we multiply the entire result by 2, to bring it closer to the original normalized range (Figure 4.28).

Here is how it is implemented in code:

```
powder_sugar_effect  = 1.0 -  exp( - light_samples * 2.0 );
beers_law = exp( - light_samples );
light_energy = 2.0 * beers_law * powder_sugar_effect;
```

Some results both from an isolated test case and from our solution in-game are shown in Figure 4.29.

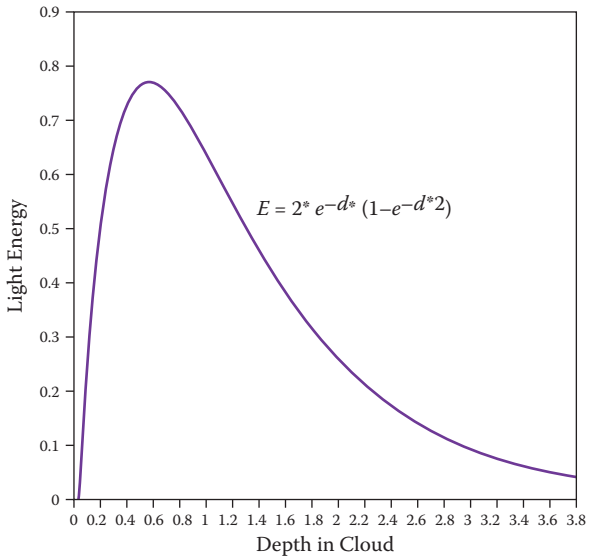


Figure 4.28. The combined Beer's-Powder function.

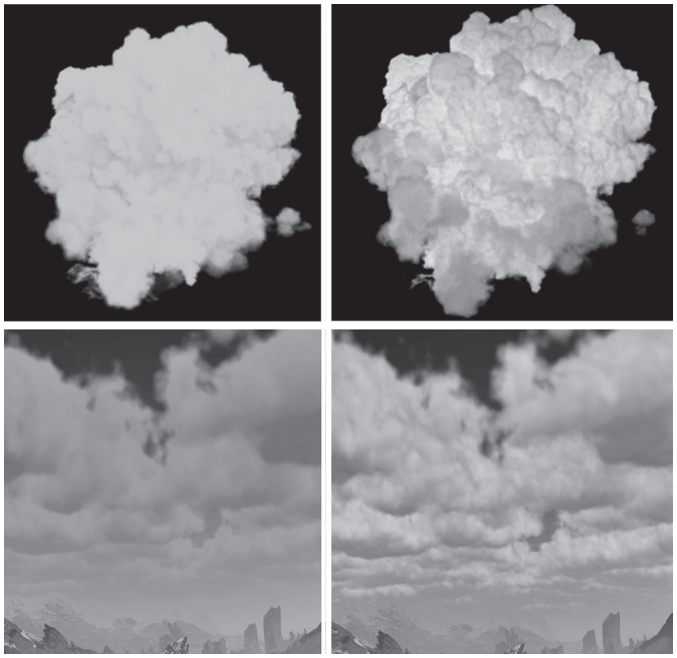


Figure 4.29. Lighting model without our function (top left) and with our function (top right). In-game results without our function (bottom left) and with our function (top right).

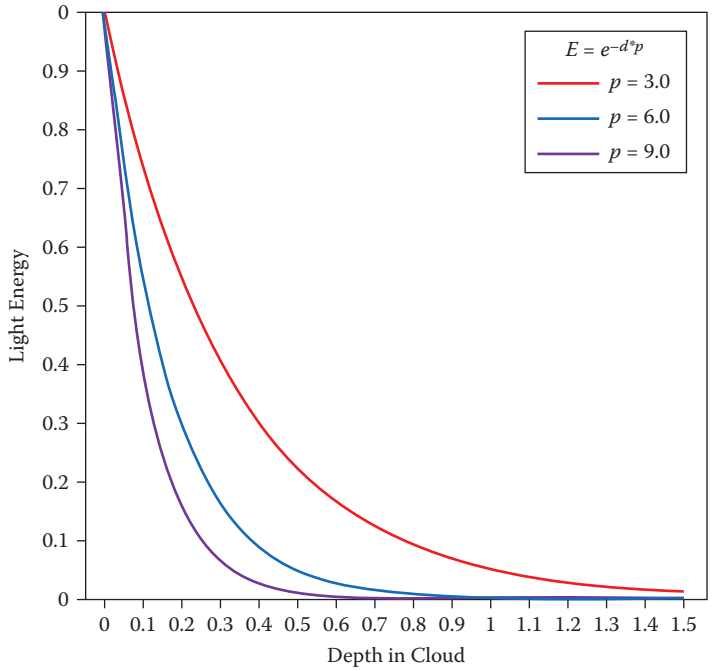


Figure 4.30. Several results using different absorption levels.

4.4.5 Rain Clouds

Our system also models the darker bases of rain clouds. Rain clouds are darker than other low-level clouds because the water droplets have condensed so much that most of the light gets absorbed before reaching our eye.

So, since we already have a precipitation attribute for the point that we are sampling, we can use it to artificially "thicken" the cloud material. This task is easily accomplished by increasing the sampled density that goes into the Beer's-Powder function; see Figure 4.30. The variable p stands for precipitation.

Figure 4.31 shows some results.

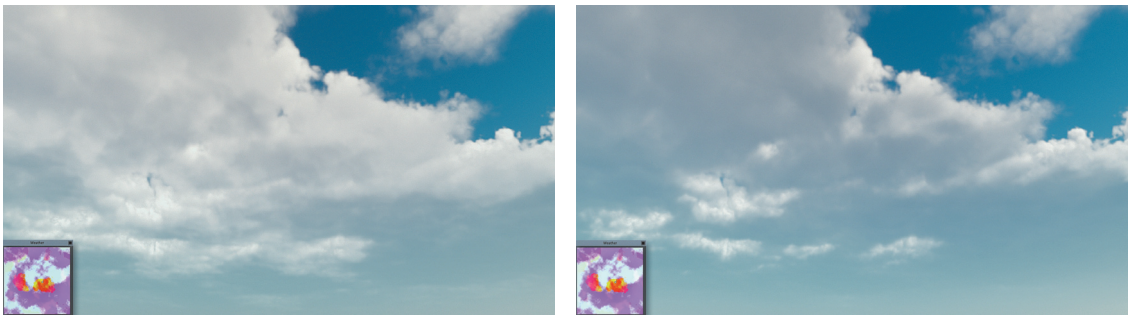


Figure 4.31. Rain clouds with (left) and without (right) increased light absorption.

4.4.6 Lighting Model Summary

In review, our lighting model is a combination of four components:

1. Beer's law (August Beer, 1852),
2. Henyey-Greenstein phase function (Henyey and Greenstein, 1941),
3. in-scattering probability function (powdered sugar effect),
4. rain cloud absorption gain.

With E as light energy, d as the density sampled for lighting, p as the absorption multiplier for rain, g as our eccentricity in light direction, and θ as the angle between the view and light rays, we can describe our lighting model in full:

$$E = 2.0 \times e^{-dp} \times \left(1 - e^{-2d}\right) \times \frac{1}{4\pi} \frac{1 - g^2}{1 + g^2 - 2g\cos(\theta)^{3/2}}.$$

4.5 Cloud Rendering

Choosing where to sample data to build the image is very important for performance and image quality. Our approach tries to limit expensive work to situations where it could potentially be required.

4.5.1 Spherical Atmosphere

The first part of rendering with a ray march is deciding where to start. When the viewer is located on a seemingly "flat" surface such as the ocean, the curvature of the Earth clearly causes clouds to descend into the horizon. This is because the Earth is round and cloud layers are spherical rather than planar. (See Figure 4.32.)

In order to reproduce this feature, our ray march takes place in a 3.5 km thick spherical shell starting at 1.5 km above the surface of the Earth. We use a sphere intersection test to determine the start and end points for our ray march. As we look toward the horizon, the ray length increases considerably, which requires that we increase the number of potential samples. Directly above the player, we take as many as 64 steps and at the horizon we take as many as 128 steps. There are several optimizations in our ray-march loop, allowing it to exit early, so the average sample count is much lower than this.

4.5.2 Ray March Optimizations

Instead of evaluating the full cloud density function every time, we only evaluate the low-frequency part of the cloud density function until we are close to a cloud. Recall that our density function uses low-detail Perlin-Worley noise to establish

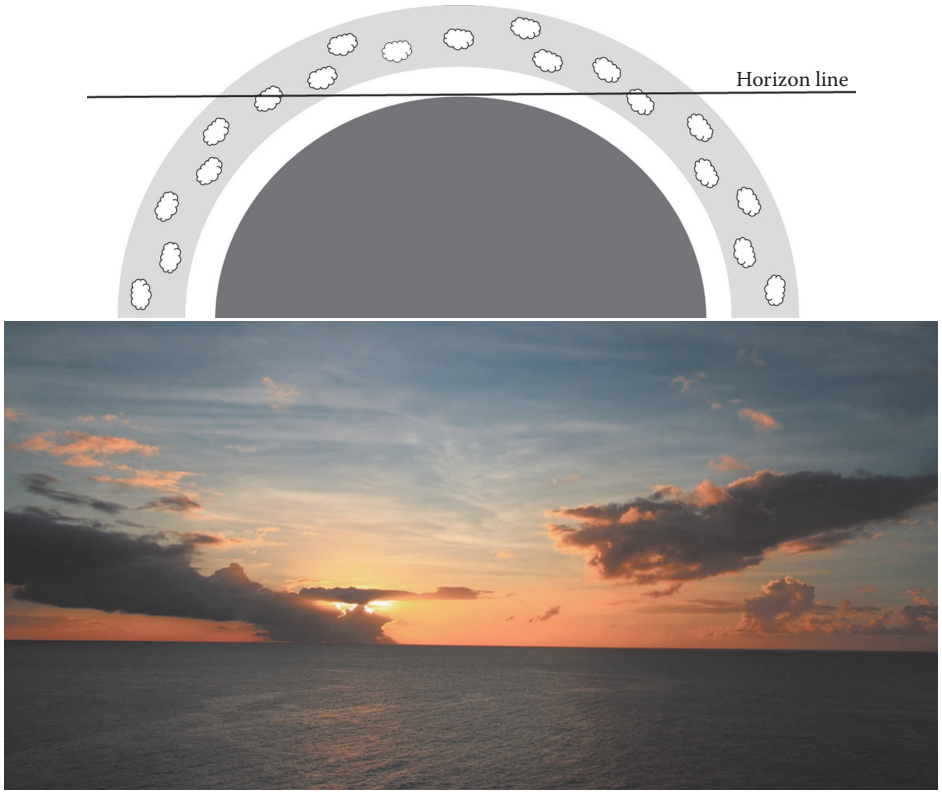


Figure 4.32. Spherical atmosphere.

the base shape of our clouds and higher frequencies of Worley noise, which it applies as an erosion from the edges of this base cloud shape. Evaluating just the low-frequency part of the density function means one 3D texture is read instead of two, which is a substantial bandwidth and instruction count savings. Figure 4.33 illustrates the step through empty air using "cheap" samples and then the switch to expensive samples when close to a cloud. Once several samples return zero, we return to the "cheap" sample.

To implement this in code, we start with a `cloud_test` value of zero and accumulate density in a loop using a boolean value of `true` for our sampler. As long as the `cloud_test` is 0.0, we continue on our march searching for the cloud boundary. Once we get a nonzero value, we suppress the march integration for that step and proceed using the full cloud density sample. After six consecutive full cloud density samples that return 0.0, we switch back to the cloud boundary search. These steps ensure that we have exited the cloud boundary and do not trigger extra work.

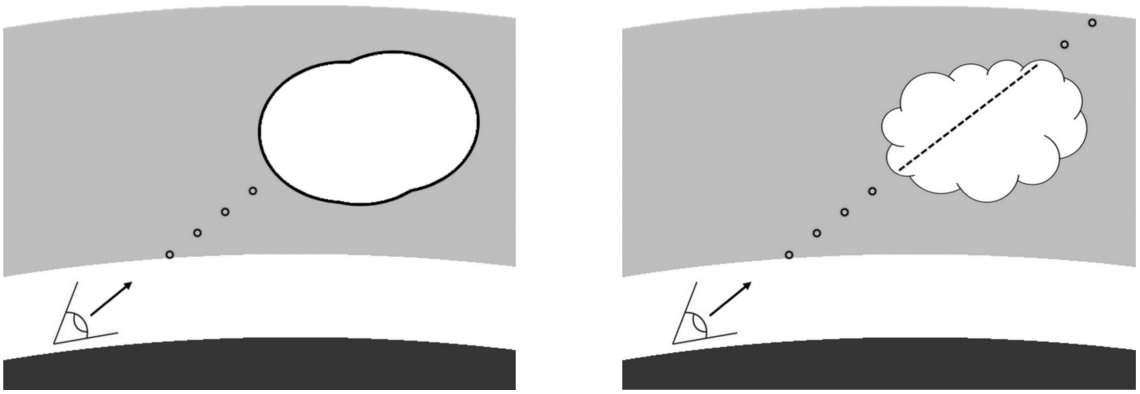


Figure 4.33. Cloud boundary detection (left), and full samples inside of the cloud boundary (right).

```
float density = 0.0;
float cloud_test = 0.0;
int zero_density_sample_count = 0;

// Start the main ray-march loop.
for (int i = 0; i <sample_count; i++)
{
    // cloud_test starts as zero so we always evaluate the
    // second case from the beginning.
    if(cloud_test > 0.0)
    {
        // Sample density the expensive way by setting the
        // last parameter to false, indicating a full sample.
        float sampled_density = SampleCloudDensity(p,
             weather_data, mip_level, false);

         // If we just samples a zero, increment the counter.
         if( sampled_density = 0.0)
        {
            zero_density_sample_count ++;
        }
       // If we are doing an expensive sample that is still
       // potentially in the cloud:
        if(zero_density_sample_count != 6)
        {
            density += sampled_density;
            p += step;
        } // If not, then set cloud_test to zero so that we go
          // back to the cheap sample case.
        else
        {
            cloud_test = 0.0;
            zero_density_sample_count = 0;
        }
    }
    else
    {
        // Sample density the cheap way, only using the
        // low-frequency noise.
```

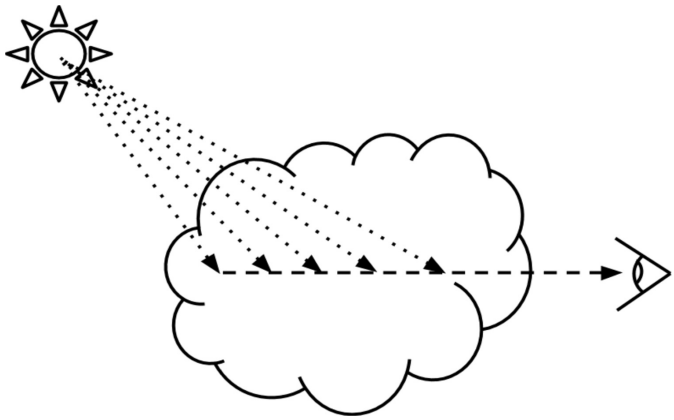


Figure 4.34. A light ray march for each view ray-march step.

```
            cloud_test = SampleCloudDensity(p, weather_data,
                                            mip_level, true);
            if( cloud_test == 0.0)
            {
                p += step;
            }
        }
    }
```

This algorithm cuts the number of 3D texture calls in half for the best case, where we are marching through empty sky.

To calculate the lighting, more samples need to be taken toward the light at each ray-march step. The sum of these samples is used in the lighting model and then attenuated by the current sum of density along the view ray for each view ray-march step. Figure 4.34 illustrates a basic light sample integration march within a ray march.

Because we are targeting for use in a game engine that is supporting many other GPU intensive tasks, we are limited to no more than six samples per ray-march step.

One way to reduce the number of light samples is to execute them only when the ray march steps inside of a cloud. This is an important optimization because light samples are extremely costly. There is no change in the visual result with this optimization.

```
...
            density += sampled_density;
            if( sampled_density != 0.0)
            {
                // SampleCloudDensityAlongRay just walks in the
                // given direction from the start point and takes
```

```
                // X number of lighting samples.
                density_along_light_ray =
                    SampleCloudDensityAlongRay(p)
            }
            p += step;
...
```

4.5.3 Cone-Sampled Lighting

The obvious way to find the amount of sun illumination is by measuring the transmittance of the cloud between the query point and the sun. However, the light at any point in a cloud is greatly affected by the light in regions around it in the direction of the light source. Think of it as a funnel of light energy that culminates at our sample position. To make sure that the Beer's law portion of our lighting model is being influenced in this way, we take our six light samples in a cone that spreads out toward the light source, thus weighting the Beer's law attenuation function by including neighboring regions of the cloud. See Figure 4.35.

Banding artifacts present themselves immediately because of the low number of samples. The cone sampling helps break up the banding a bit, but to smooth it out further, we sample our densities at a lower mip level.

To calculate the cone offset, we used a kernel of six noise results between $-(1,1,1)$ and $+(1,1,1)$ and gradually increased its magnitude as we march away from our sample position. If the accumulated density along the view march has surpassed a threshold value where its light contribution can be more generalized (we used 0.3), we switch our samples to the low-detail mode to further optimize the light march. There is very little visual difference at this threshold.

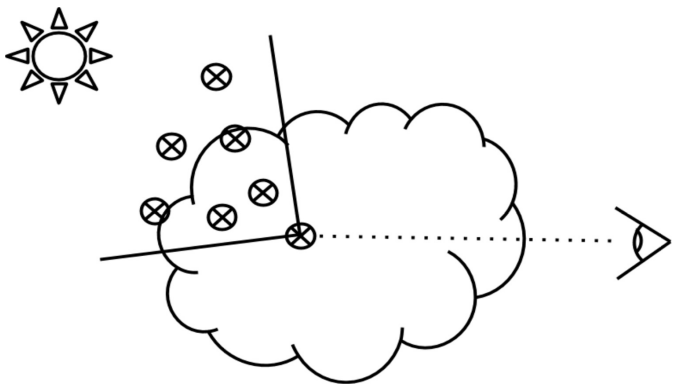


Figure 4.35. A cone-shaped sample area for the light ray-march samples.

```
static float3 noise_kernel[] =
{
    some noise vectors
}

// How wide to make the cone.
float cone_spread_multplier = length(light_step);

// A function to gather density in a cone for use with
// lighting clouds.
float SampleCloudDensityAlongCone(p, ray_direction)
{
    float density_along_cone = 0.0;

    // Lighting ray-march loop.
    for(int i=0; i<=6; i++)
    {
        // Add the current step offset to the sample position.
        p += light_step + ( cone_spread_multiplier *
                            noise_kernel[i] * float(i) );
        if( density_along_view_cone < 0.3)
        {
            // Sample cloud density the expensive way.
            density_along_cone += SampleCloudDensity(p,
                        weather_data, mip_level + 1, false);
        }
        else
        {
           // Sample cloud density the cheap way, using only
           // one level of noise.
            density_along_cone += SampleCloudDensity(p,
                        weather_data, mip_level + 1, true);
        }
    }
}
```

Additionally, to account for shadows cast from distant clouds onto the part of the cloud for which we are calculating lighting, we take one long distance sample at a distance of three times the length of the cone. (See Figure 4.36.)

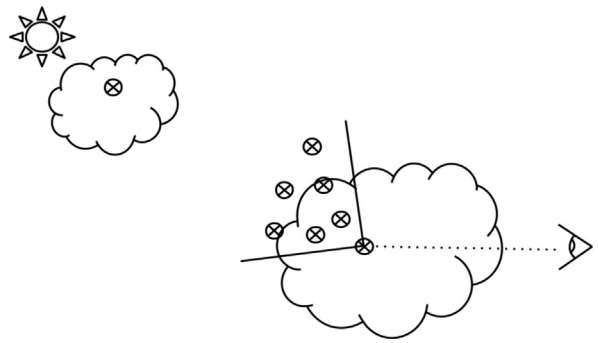


Figure 4.36. Long distance light sample combined with the cone samples.

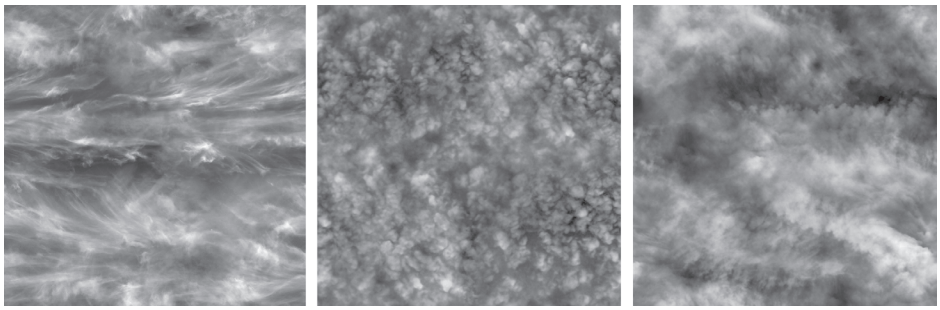


Figure 4.37. Several alto and cirrus cloud textures used instead of a ray march.

4.5.4 High Altitude Clouds

Our approach only renders low-level clouds volumetrically. High-altitude clouds are represented with scrolling textures. However, in order to integrate them with the volumetric clouds, they are sampled at the end of the ray march. The cost of this texture read is negligible for a 512^2 texture with three channels. We animate them in a wind direction that is different from the wind direction in our weather system to simulate different wind directions in different cloud layers. (See Figure 4.37.)

4.5.5 Results

A sequence of lighting results that illustrates a changing time of day is illustrated in Figure 4.38.

4.6 Conclusion and Future Work

This approach produces realistic, evolving cloudscapes for all times of day and completely replaces our asset-based approaches to clouds. It also means that the memory usage for our entire sky is limited to the cost of a few textures that total 20 MB instead of hundreds of megabytes for multiple sky domes and billboards for varying weather conditions and times of day. Performance on the GPU is roughly 20 ms, but when we build our image using temporal reprojection, that number reduces to 2 ms [Schneider 15, slide 91–93].

The in-scattering probability function was based on a thought experiment, but we are researching this further. We plan to use the brute-force approach used by Magnus Wrenninge [Wrenninge 15], which produces the dark edges naturally, to gather data points along the light ray and develop a function that fits these data more precisely.

Figure 4.38. Time lapse of a cloudscape, captured from the Playstation 4.

4.7 Acknowledgments

I would like to thank Nathan Vos, Michal Valient, Elco Vossers, and Hugh Malan for assistance with our development challenges. I would also like to thank Jan-Bart van Beek, Marijn Giesbertz, and Maarten van der Gaag for their assistance in accomplishing our look goals for this project.

Additionally, I would like to personally thank colleagues whose work has greatly influenced this: Trevor Thomson, Matthew Wilson, and Magnus Wrenninge.

Bibliography

[Beer 52] A. Beer. "Bestimmung der Absorption des rothen Lichts in farbigen Flüssigkeiten" (Determination of the Absorption of Red Light in Colored Liquids). *Annalen der Physik und Chemie* 86 (1852), 78–88.

[Clausse and Facy 61] R. Clausse and L. Facy. *The Clouds*. London: Evergreen Books, LTD., 1961.

[Henyey and Greenstein 41] L. G. Henyey and J. L. Greenstein. "Diffuse Radiation in the Galaxy." *Astrophysical Journal* 93 (1941), pp. 78–83.

[Mandelbrot and van Ness 68] B. Mandelbrot and J. W. van Ness. "Fractional Brownian Motions, Fractional Noises and Applications." *SIAM Review* 10:4 (1968), 422–437.

[Perlin 85] K. Perlin. "An Image Synthesizer." In *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 287–296. New York: ACM Press, 1985.

[Pharr and Humphreys 10] M. Pharr and G. Humphreys. *Physically Based Rendering: From Theory to Implementation.* Boston: Morgan Kaufmann, 2010.

[Quilez 13] I. Quilez. "Clouds." *Shadertoy.com*, https://www.shadertoy.com/view/xslgrr, 2013.

[Schneider 15] A. Schneider. "The Real-Time Volumetric Cloudscapes Of Horizon: Zero Dawn." Paper presented at ACM SIGGRAPH, Los Angeles, CA, August 26, 2015.

[Simul 13] Simul. "TrueSKY." http://simul.co/truesky/, 2013.

[Van De Hulst 57] H. Van De Hulst. *Light Scattering by Small Particles*. New York: Dover Publications, 1957.

[Worley 96] Steven S. Worley. "A Cellular Texture Basis Function." In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, pp. 291–294. New York: ACM Press, 1996.

[Wrenninge 13] M. Wrenninge. *Production Volume Rendering: Design and Implementation*. Boca Raton, FL: CRC Press, 2013.

[Wrenninge 15] M. Wrenninge. "Art-Directable Multiple Volumetric Scattering." In *ACM SIGGRAPH 2015 Talks*, article no. 24. New York: ACM Press, 2015.