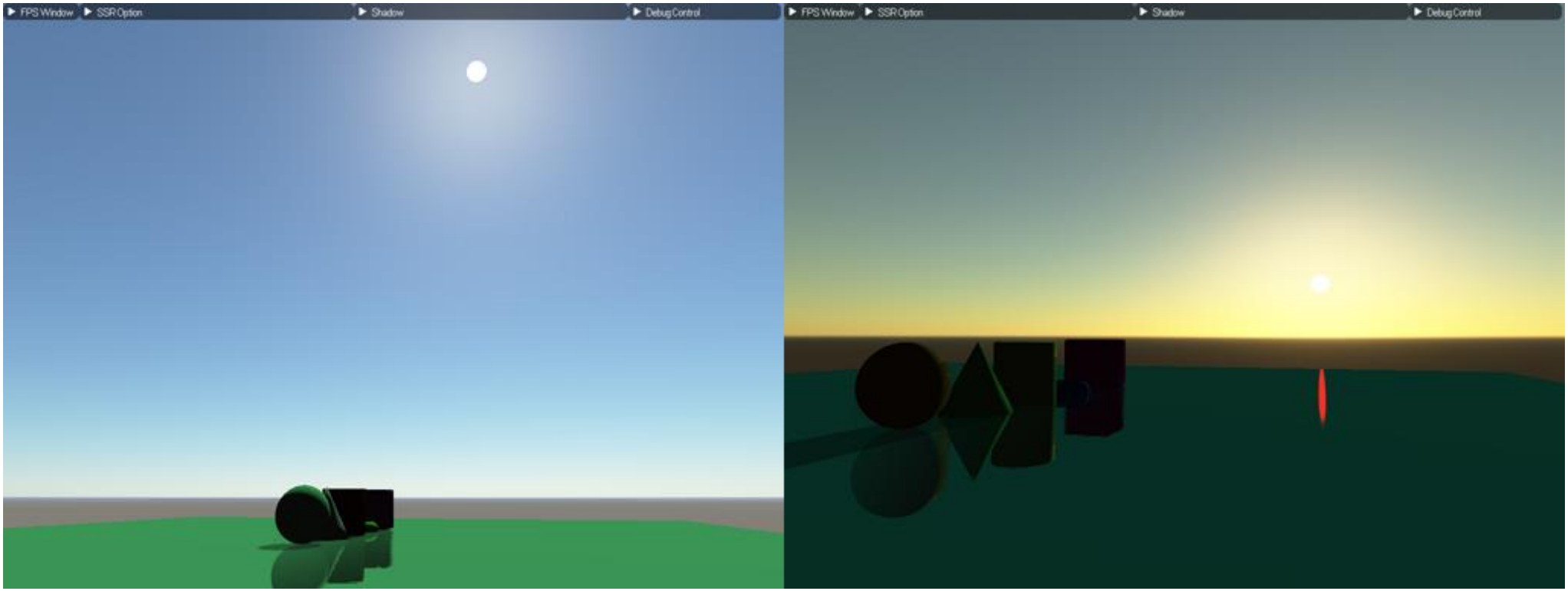
Precomputed Atmospheric Scattering



Precomputed Atmospheric Scattering

Eric Bruneton, Fabrice Neyret의 2008 논문. [Bruneton08]

지구의 대기 렌더링에 필요한 데이터(투과율, 복사 조도, 산란)를 **텍스처 형태로 미리 계산** (Precomputed) 하고 이를 이용하여 실시간으로 대기를 렌더링 하는 방법이다.

UE4의 Atmospheric Fog가 이 모델을 그대로 구현하였다. (파일명이 Atmospheric~ 으로 시작하는 파일들)

저자가 OpenGL로 작성된 소스 코드를 공개하였기 때문에 구현하는 건 어렵지 않다.
UE4 코드도 샘플 소스와 거의 유사하다. (Precomputed 부분은 변수 명 빼고 완전 동일)

문제는 코드를 보고 이해하는 것. 대기 산란에 대한 배경 지식이 필요했다.

이 ppt는 해당 논문을 이해하기 위한 배경 지식과 소스 코드를 분석한 결과를 정리하기 위하여 작성하게 되었다.

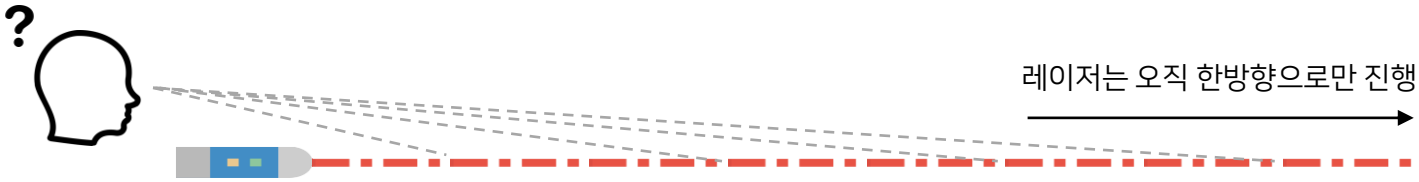
빛의 산란(Scattering)

다음과 같은 상황을 생각해보자.

손에 레이저 포인터를 들고 자신의 정면방향으로 레이저를 쏘는 상황이다.

Q. 이때 레이저의 빛줄기를 볼 수 있을까?

A. 볼 수 없다.



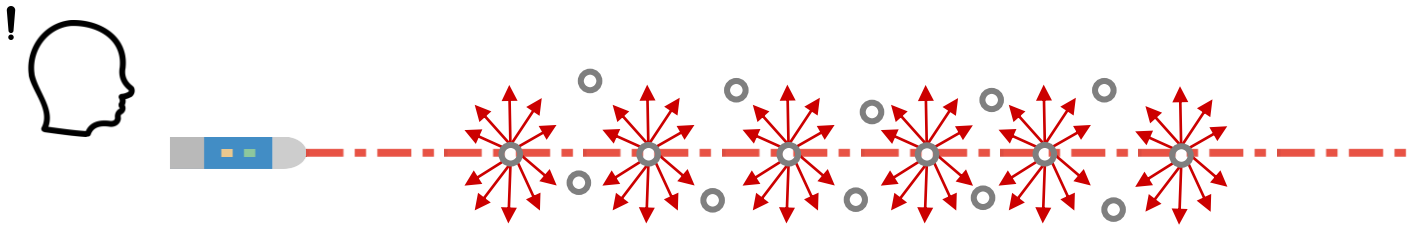
레이저의 그 어떤 빛도 눈으로 들어올 수 없다.

Q. 그럼 레이저의 빛줄기를 볼 수 있는 방법은?

빛의 산란(Scattering)

A. 레이저의 빛이 방출되는 경로에 모래를 끼얹는다.

https://youtu.be/qybisPmc2xw?t=67



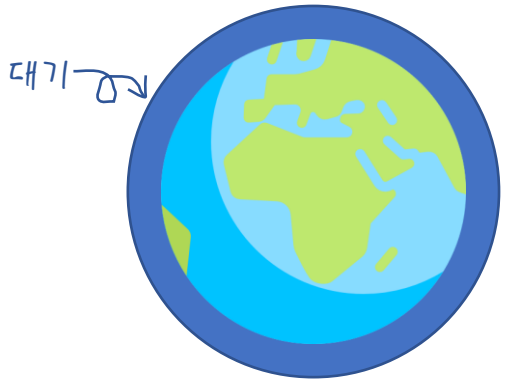
모래 알갱이에 레이저 빛이 부딪히면 온사방으로 퍼져 눈으로 빛이 들어오게 된다.

이렇게 빛이 어떤 **입자**들과 부딪혀서 사방으로 흩어지는 현상을 산란(Scattering) 이라 한다.

낮에 푸른 하늘을 볼 수 있는 것도 산란 때문이다.

대기(Atmosphere)

천체의 주위를 둘러싸고 있는 기체층이다.



지구의 대기는 다음과 같은 물질로 구성되어 있다.

종류	부피
질소 (N_2)	78.084%
산소 (O_2)	20.946%
아르곤 (Ar)	0.9340%
이산화탄소 (CO_2)	0.04%
네온 (Ne)	0.001818%
헬륨 (He)	0.000524%
메탄 (CH_4)	0.000179%
수증기 (H_2O)	0.001% ~ 5%

레이저 빛이 모래에 부딪혀 여러 방향으로 흩어지듯이 태양빛은 지구에 도달할 때 까지 대기의 여러 분자와 부딪혀 산란된다.

대기 산란(Atmospheric Scattering)

대기의 산란은 입자의 크기에 따라서 크게 2가지 이론으로 설명할 수 있다.

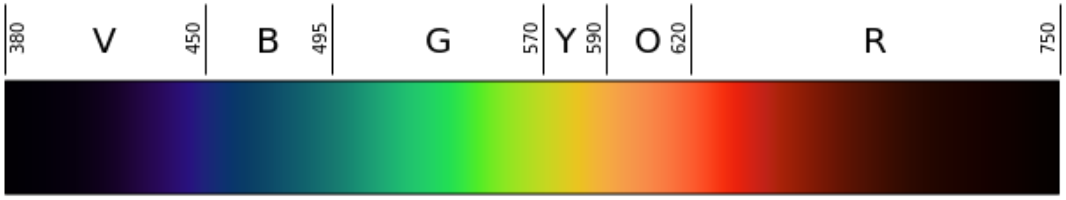
첫번째는 **입자의 크기가 전자기파의 파장보다 작을 경우**이다. (Molecules)

대기를 이루는 입자(주로 질소, 산소)는 가시광선 빛의 파장크기 보다 크기가 훨씬 작은데 이 때 빛의 산란을 **레일리 산란(Rayleigh Scattering)**이라 한다.

레일리 산란은 빛의 파장의 영향을 크게 받는데 빛의 파장의 4 제곱에 반비례하여 산란되는 빛의 양이 줄어든다.

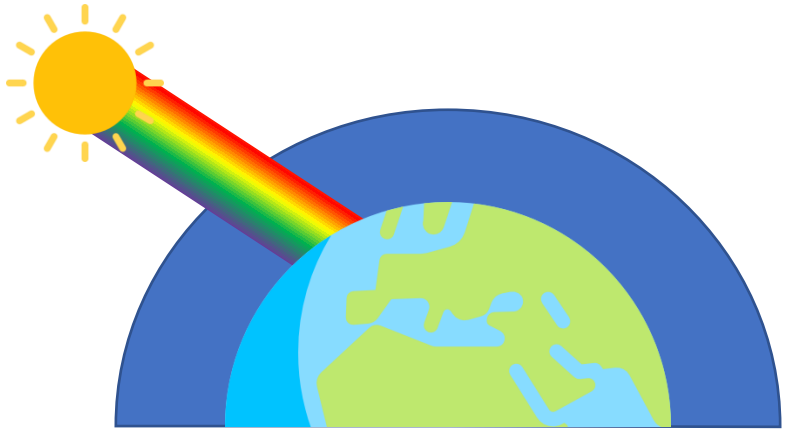
즉, 파장이 길 수록 산란이 덜 일어나게 된다는 것이다.

눈에 보이는 가시광선의 파장은 다음과 같은데 파랑색에서 붉은색으로 갈 수록 파장이 길어지고 산란이 덜 일어나게 된다.



대기 산란(Atmospheric Scattering)

하늘이 푸르게 보이는 이유는 레일리 산란에 의해 파란색 빛이 더 많이 산란되어 눈으로 들어오기 때문이다.



해질녘에 하늘이 붉은 이유도 레일리 산란 때문인데 태양의 고도가 낮아지면 통과해야 하는 대기층이 길어져 산란되기 쉬운 파란색 빛은 산란되어 버리고 산란되기 어려운 붉은 빛만 눈으로 들어오기 때문이다.

대기 산란(Atmospheric Scattering)

두번째는 **입자의 크기가 전자기파의 파장과 비슷할 경우**이다. (Aerosol)

수증기나 매연, 얼음 알갱이 등은 입자의 크기가 빛의 파장과 비슷한데 이 때 빛의 산란을 **미 산란(Mie Scattering)** 이라 한다.

미 산란의 경우 빛의 파장보다는 입자의 밀도, 크기, 모양에 영향을 받는다.

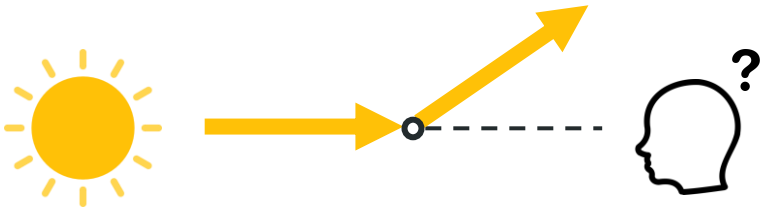
구름을 이루는 물 입자나 얼음 입자는 미 산란이 일어날 만큼 충분히 크기 때문에 구름이 희게 보인다. (특정 빛 파장에 상관없이 태양 빛을 그대로 산란 시키기 때문)

정리하자면 빛이 대기를 이루는 입자와 충돌하여 산란되어 하늘이 우주처럼 검게 보이지 않는 것이다. 하지만 입자와 충돌하였다고 반드시 빛을 볼 수 있는 것은 아니다.

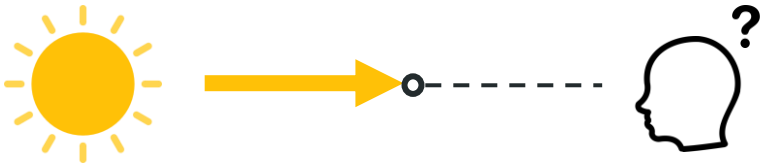
입자와 빛

부딪힌 입자가 없어 빛을 보았다.

빛이 입자에 부딪혀 빛을 못 봤다.

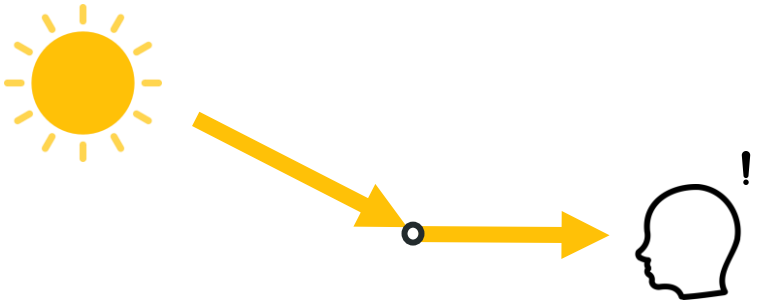


Out-Scattering



Absorption

빛이 입자에 부딪힌 덕분에 빛을 보았다.



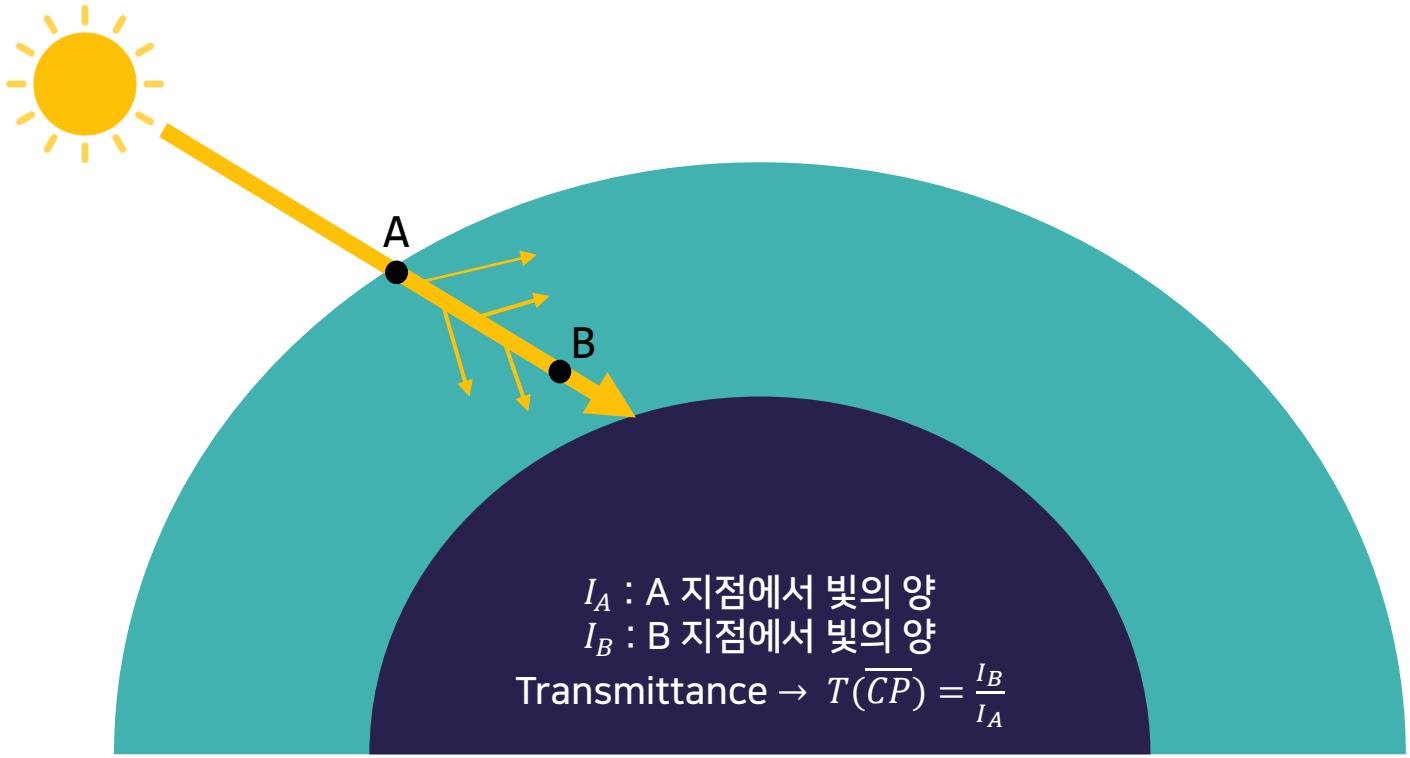
In-Scattering

Transmittance (투과율)

산란으로 항상 빛을 볼 수 있는 것은 아니다.

입자와 부딪혀 눈으로 들어오지 않는 경우도 있다. (Out-Scattering, Absorption)

투과율은 **특정 파장의 입사광이 물질을 통과하는 비율**로 가장 먼저 계산하는 정보이다.

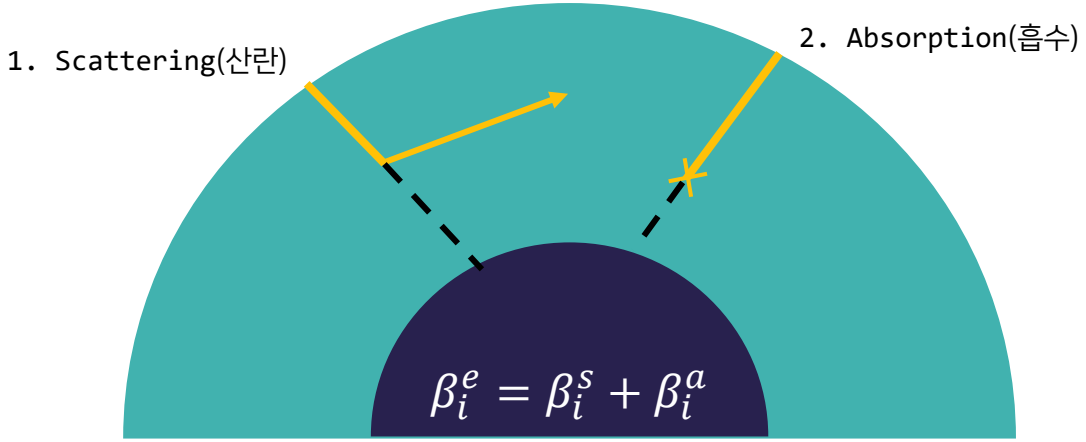


Transmittance (투과율)

$$\mathrm{T}(\mathrm{x}, \mathrm{x}_0) = e^{-\int_x^{x_0} \Sigma_{i\in\{R,M\}} \beta_i^e(y)dy}$$

β_i^e = extinction coefficient (소광 계수)

: 빛이 1m를 진행하는 동안 물질이 빛을 소멸시키는 정도를 나타내는 계수 (= 확률)



즉, 빛이 매질에 의해서 진행방향을 바꾸거나[1] 흡수되어[2] 본래의 방향으로 진행하지 못하는 정도

x, x_0 = 빛의 진행 구간

R = Rayleigh (레일리)

M = Mie (미)

y = 고도

Scattering Equation (산란 방정식)

소광 계수(β_i^e)는 산란 방정식(scattering equation)에서 유도된다.

레일리 산란 방정식 $S(\lambda, \theta, h)$는 다음과 같다.

$$S(\lambda, \theta, h) = \frac{\pi^2 (n^2 - 1)^2}{2} \frac{\rho(h)}{N} \frac{1}{\lambda^4} (1 + \cos^2 \theta)$$

λ : 파장.

θ : 산란 각도.

h : 고도.

$n = 1.00029$: 공기의 굴절률.

$N = 2.504 * 10^{25}$: 표준 대기의 분자 개수 밀도. (= m^3 당 분자의 수)

$\rho(h)$: 밀도 비.(density ratio) 해수면에서는 1이며 고도가 높아질 수록 기하급수적으로 감소한다.

※ 여기서 사용되는 레일리 산란 방정식은 Display of The Earth Taking into Account Atmospheric Scattering 논문에서 유래한 식이다. 위키피디아에 소개된 것 과 매우 다르다.

Scattering Equation (산란 방정식)

레일리 산란 방정식 $S(\lambda, \theta, h)$은 특정 방향(θ)으로 얼마나 많은 빛이 산란되는지 보여준다.

방향에 상관없이 빛이 산란되는 총량을 구하려면 모든 방향을 고려할 필요가 있다.

$$\beta(\lambda, h) = \int_S S(\lambda, \theta, h) d\omega$$

$$= \int_0^{2\pi} \int_0^{\pi} S(\lambda, \theta, h) sin\theta d\theta d\phi$$

$$= \int_0^{2\pi} \int_0^{\pi} \frac{\pi^2 (n^2 - 1)^2}{2} \frac{\rho(h)}{N} \frac{1}{\lambda^4} (1 + \cos^2 \theta) sin\theta d\theta d\phi$$

$$= \frac{\pi^2 (n^2 - 1)^2}{2} \frac{\rho(h)}{N} \frac{1}{\lambda^4} \int_0^{2\pi} \int_0^{\pi} (1 + \cos^2 \theta) sin\theta d\theta d\phi$$

$$= \frac{\pi^2 (n^2 - 1)^2}{2} \frac{\rho(h)}{N} \frac{1}{\lambda^4} \int_0^{2\pi} \frac{8}{3} d\phi$$

$$= \frac{\pi^2 (n^2 - 1)^2}{2} \frac{\rho(h)}{N} \frac{1}{\lambda^4} \frac{16\pi}{3}$$

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

Scattering Coefficient (산란 계수)

여기서 계산한 $\beta(\lambda, h)$를 레일리 산란 계수(Rayleigh Scattering Coefficient)라 하고 β_R^s 로 표기한다.

그리고 레일리 산란의 경우 산란계수가 곧 소광 계수이기 때문에 $\beta_R^e = \beta_R^s$이 된다.

미 산란의 경우 산란 계수 β_M^s 는 $\beta_M^s(h, \lambda) = \beta_M^s(0, \lambda)\rho(h)$가 되는데 레일리 산란과는 달리 소광 계수 β_M^e는 산란 계수(β_M^s)와 흡수 계수(β_M^a)를 합해서 얻을 수 있다.

$$\beta_M^e = \beta_M^s + \beta_M^a$$

주목해야 할 점은 두 산란 계수 모두 파장을 정하면 밀도 비($\rho(h)$)를 제외하고는 상수가 된다는 점이다.

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

$$\beta_M^s(h) = \beta_M^s(0, \lambda)\rho(h)$$

즉 밀도 비에 대한 단항식이라고 생각할 수 있다.

Density ratio (밀도 비)

산란의 강도는 대기 밀도에 비례한다. 평방 미터 당 분자가 많으면 산란하기 쉽다는 사실은 쉽게 추측할 수 있다.

밀도 비 $\rho(h)$는 고도 h에서의 밀도와 고도 0에서의 밀도간 비율로 다음과 같이 정의된다.

$$\rho(h) = \frac{density(h)}{density(0)}$$

대기의 조성은 매우 복잡하여 압력, 밀도, 온도가 다른 여러 층으로 구성되어 있는데 1976년도에 업데이트 된 미국 표준 대기의 고도에 따른 밀도를 살펴보면 다음과 같다.

고도(m)	밀도(kg/m^3)
0	1.2250
500	1.1673
1000	1.1117
1500	1.0581
2000	1.0066
2500	0.95695
3000	0.90912
3500	0.86323
4000	0.81913
4500	0.77677

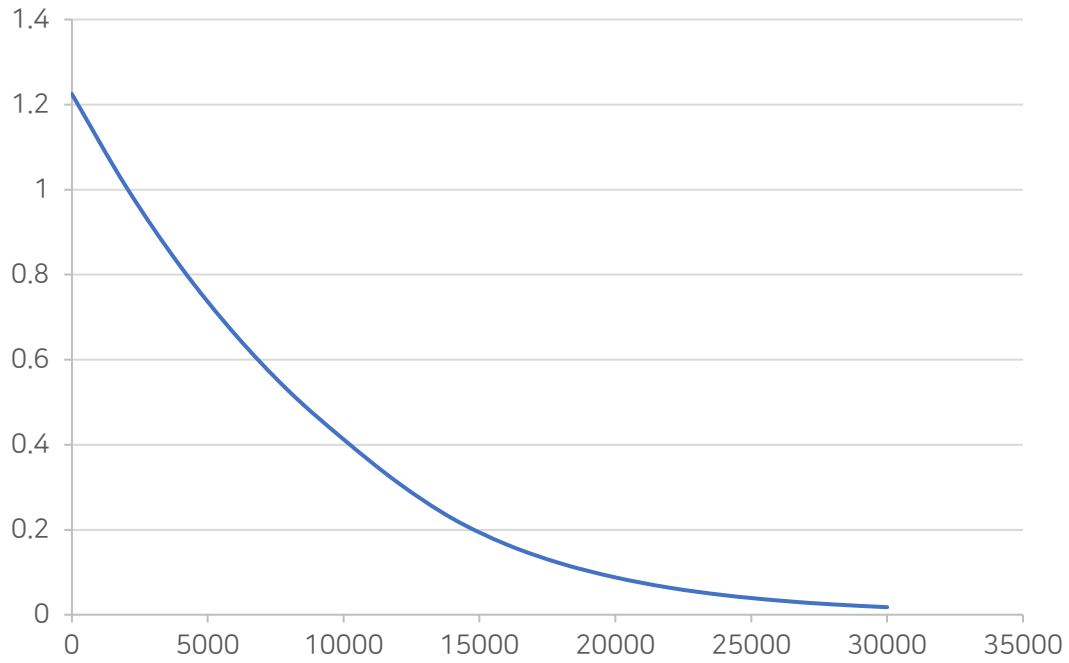
출처 : https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19770009539.pdf

Density ratio (밀도 비)

이 표를 통해 밀도 비를 구해보면 고도가 0 일 때는 1, 고도가 4000인 경우 0.66868이 된다.

소광 계수를 계산할 때는 위와 같은 표를 사용하여 밀도 비를 계산하는 것이 아니라 **근사식을 사용**하게 된다.

고도에 따른 밀도를 그래프로 그려보면 다음과 같이 지수적으로 감소하는 것을 살펴 볼 수 있다.



Density ratio (밀도 비)

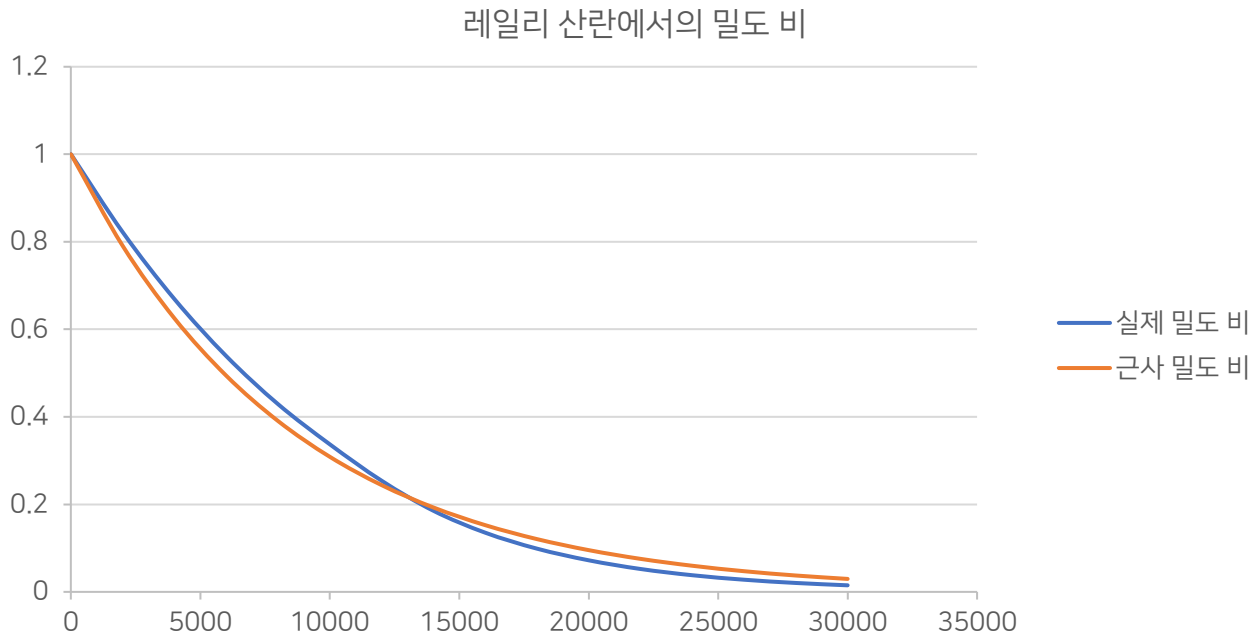
따라서 근사식도 지수 함수의 형태를 띈다. 밀도 비의 근사식은 다음과 같다.

$$\rho(h) = \mathrm{e}^{-\frac{h}{H}}$$

H : 높이 척도(scale height)

높이 척도는 어떤 양이 자연상수(e = 2.71828) 에 대해서 지수적으로 감소하는 거리로 레일리, 미 산란이 서로 다른 값을 사용한다.

산란 종류	높이 척도(km)
Rayleigh	8.5
Mie	1.2



Transmittance (투과율)

이제 투과율을 계산하는 식을 다시 살펴보면

$$\mathrm{T}(\mathrm{x}, \mathrm{x}_0) = e^{-\int_x^{x_0} \Sigma_{i\in\{R,M\}} \beta_i^e(y)dy}$$

투과율은 자연상수 e에 소광 계수 β_i^e를 지수로 갖는 식임을 알 수 있다.

자연상수 e는 어디서 나온 것일까?

투과율은 빛이 특정 구간 ($\mathrm{x} \sim \mathrm{x}_0$) 을 진행하면서 본래의 진행 방향에서 벗어나지 않는 빛의 양을 나타낸다. 따라서 원래의 빛의 세기를 I_0, 한 번의 산란 후 빛의 세기를 I_1이라 하면

$$I_1 = I_0(1 - \beta^e)$$

이다.

좀 더 근사치를 얻기 위해서 두 번의 산란을 고려한다면

$$I_1 = I_0(1 - \frac{\beta^e}{2})$$

$$I_2 = I_1\left(1 - \frac{\beta^e}{2}\right) = I_0\left(1 - \frac{\beta^e}{2}\right)^2$$

Transmittance (투과율)

여기에 무한대의 산란을 고려한다면

$$I = I_0 \lim_{n\to\infty}\left(1-\frac{\beta^e}{n}\right)^n$$

$$I = I_0 e^{-\beta^e} \qquad \because \lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n = \mathrm{e}$$

$$\frac{I}{I_0} = T = e^{-\beta^e}$$

가 되고 소광 계수가 빛이 1m 를 진행하면서 소멸될 확률이므로 임의의 거리(X)에 대한 투과율이 다음과 같이 계산된다.

$$T = e^{-\beta^e X}$$

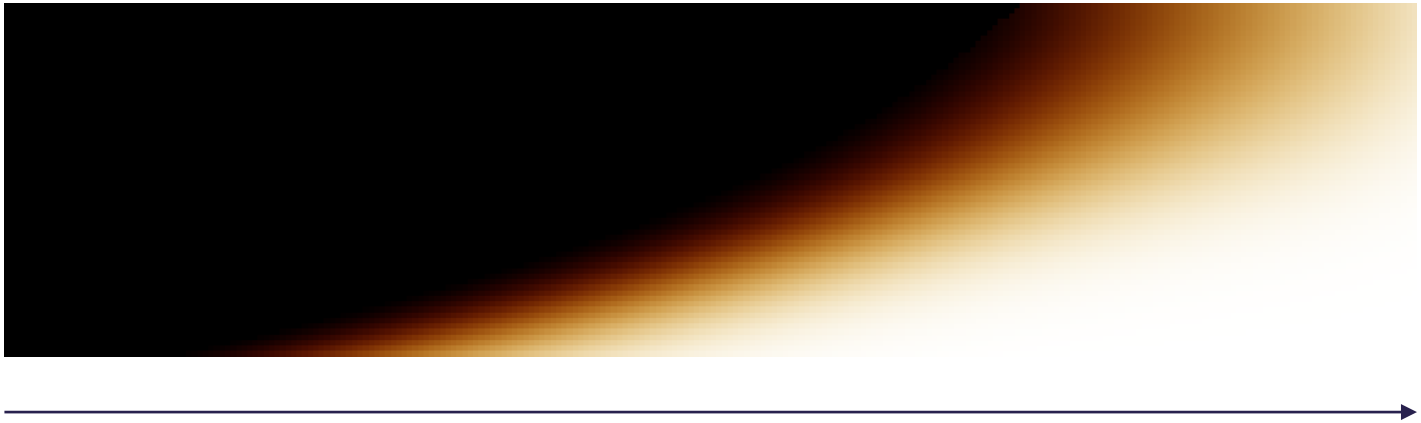
여기서 소광 계수의 상수 부분을 분리하여 이렇게 쓸 수 있는데

$$T = e^{-\beta^e_{constant}\rho(h)X}$$

$\rho(h)X$를 **광학적 깊이**(Optical depth) 라 한다.

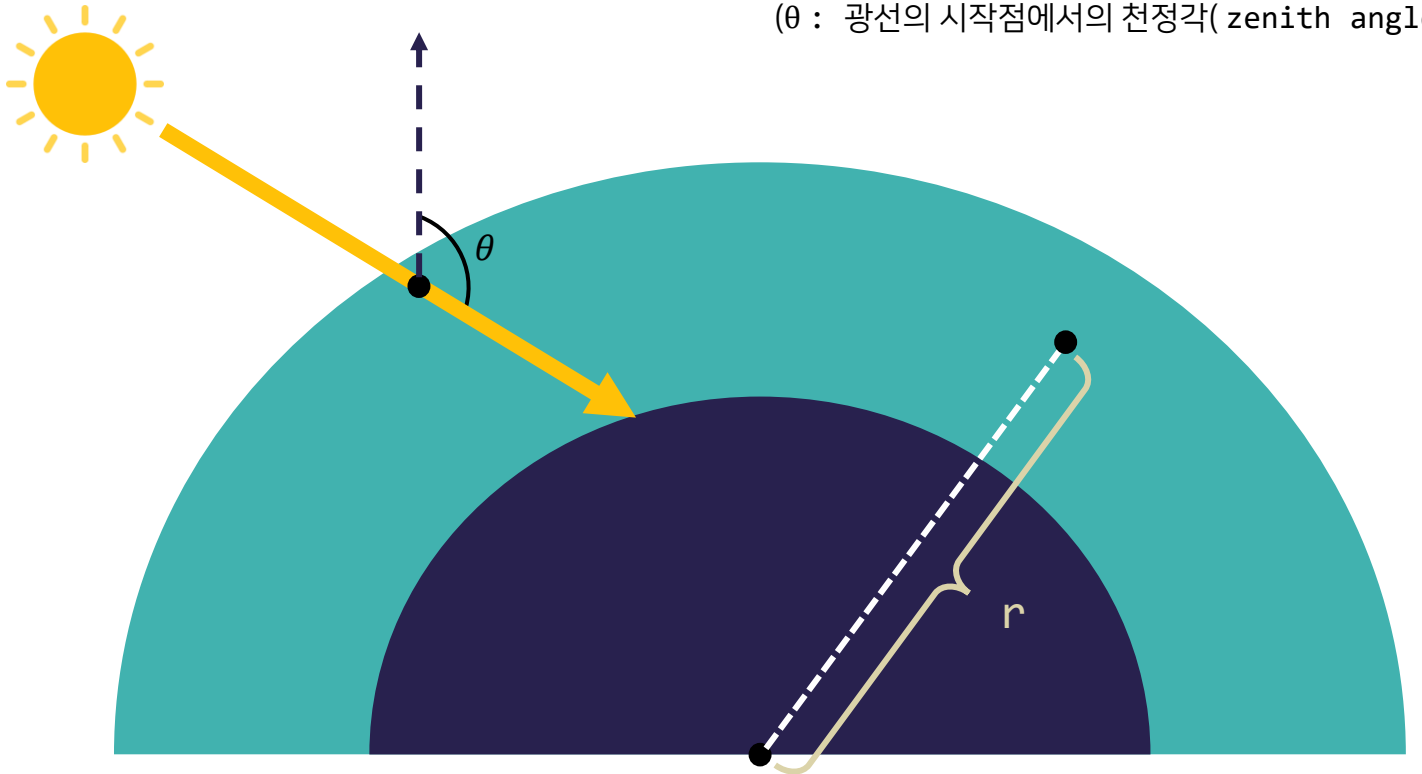
Transmittance Table

r : 고도



$\mu = cos\theta$

(θ : 광선의 시작점에서의 천정각(zenith angle))



Transmittance Table

Compute Shader를 사용하여 Transmittance Table을 계산하였다. cs_5_0의 쓰레드 제한(Maximum Threads = 1024) 때문에 적절하게 쪼개서 Dispatch 하였다.

```
[numthreads(TRANSMITTANCE_W / TRANSMITTANCE_GROUP_X, TRANSMITTANCE_H / TRANSMITTANCE_GROUP_Y, 1)]
void main( uint3 DTid : SV_DispatchThreadID )
{
	float r, muS;
	GetTransmittanceRMu( DTid, r, muS  );
	float3 depth = betaR * OpticalDepth( HR, r, muS ) + betaMEx * OpticalDepth( HM, r, muS );

	transmittanceBuffer[DTid.xy] = float4( exp( -depth ), 0.f );
}
```

쓰레드 출력 값은 $T(x, x_0) = e^{-\int_x^{x_0} \Sigma_{i\in\{R,M\}} \beta_i^e(y)dy}$을 그대로 계산하는 것을 볼 수 있다.

GetTransmittanceRMu 함수는 텍스처 UV에 맞는 고도(R)와 태양의 천정각(Mu)을 구한다.

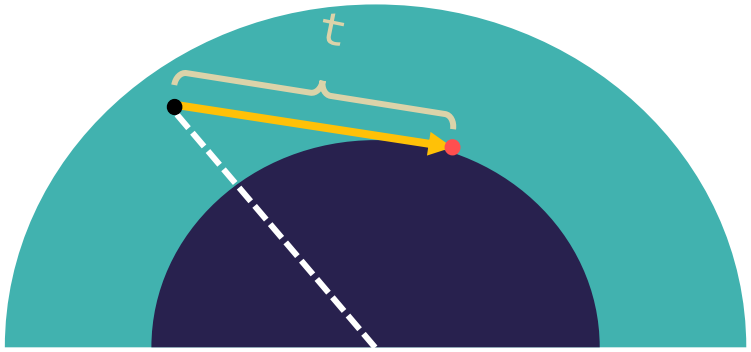
```
void GetTransmittanceRMu( uint3 DTid, out float r, out float muS )
{
	r = DTid.y / float( TRANSMITTANCE_H );
	muS = DTid.x / float( TRANSMITTANCE_W );
#ifdef TRANSMITTANCE_NON_LINEAR
	r = Rg + ( r * r ) * ( Rt - Rg );
	muS = -0.15f + tan( 1.5f * muS ) / tan( 1.5f ) * ( 1.f + 0.15f );
#else
	r = Rg + r * ( Rt - Rg );
	muS = -0.15f + muS * ( 1.f + 0.15f ); // 0 ~ ( pi/2 + epsilon )
#endif
}
```

Transmittance Table

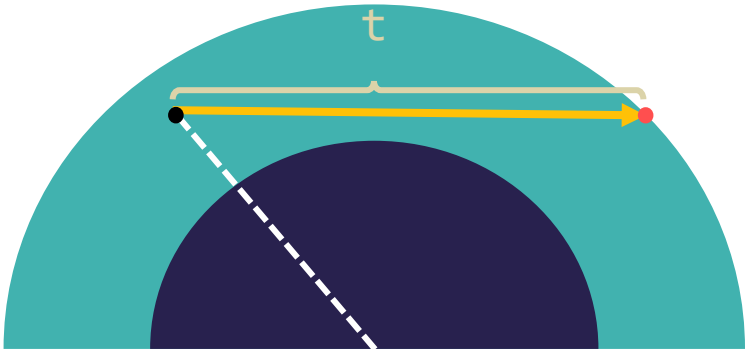
OpticalDepth는 광학적 깊이를 계산한다.

```
float OpticalDepth( float H, float r, float mu )
{
	float result = 0.f;
	float dx = limit( r, mu ) / float( TRANSMITTANCE_INTEGRAL_SAMPLES );
	float xi = 0.f;
	float yi = exp( -( r - Rg ) / H );
	for ( int i = 1; i <= TRANSMITTANCE_INTEGRAL_SAMPLES; ++i )
	{
		float xj = float( i ) * dx;
		float yj = exp( -( sqrt( r * r + xj * xj + 2.f * xj * r * mu ) - Rg ) / H ); // law of cosines
		result += ( yi + yj ) * 0.5f * dx;
		xi = xj;
		yi = yj;
	}

	return mu < -sqrt( 1.f - ( Rg / r ) * ( Rg / r ) ) ? 1e9f : result;
}
```

우선 limit 함수를 통해서 광선이 지면이나 대기의 최상단 부분에 부딪히기 까지의 진행거리를 구한다.



지면에 부딪히는 경우



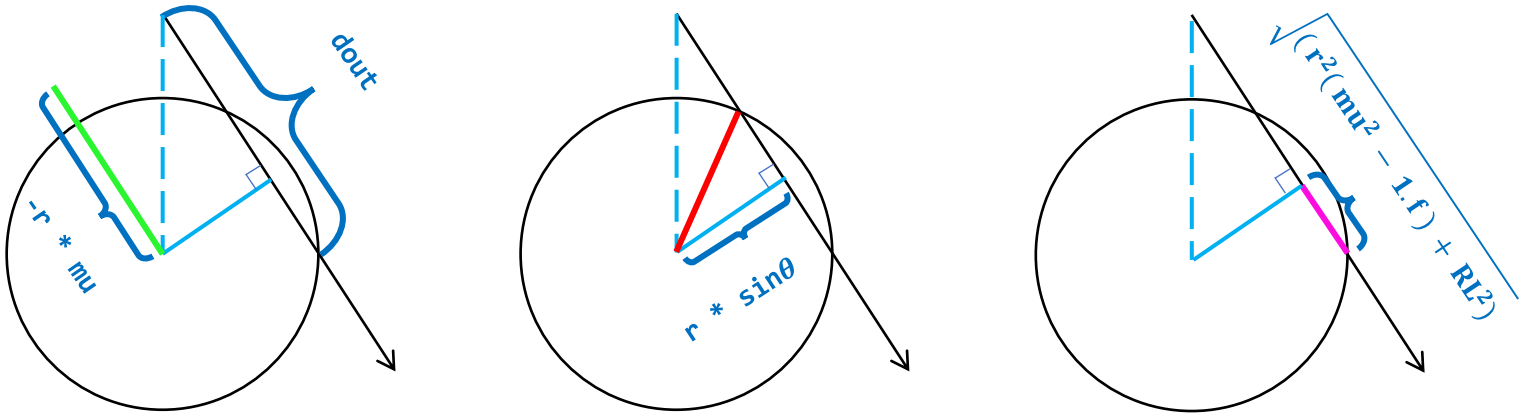
대기 최상단에 부딪히는 경우

Transmittance Table

```
float limit( float r, float mu )
{
	float dout = -r * mu + sqrt( r * r * ( mu * mu - 1.f ) + RL * RL );
	float delta2 = r * r * ( mu * mu - 1.f ) + Rg * Rg;
	if ( delta2 >= 0.f )
	{
		float din = -r * mu - sqrt( delta2 );
		if ( din > 0.f )
		{
			dout = min( dout, din );
		}
	}
	return dout;
}
```

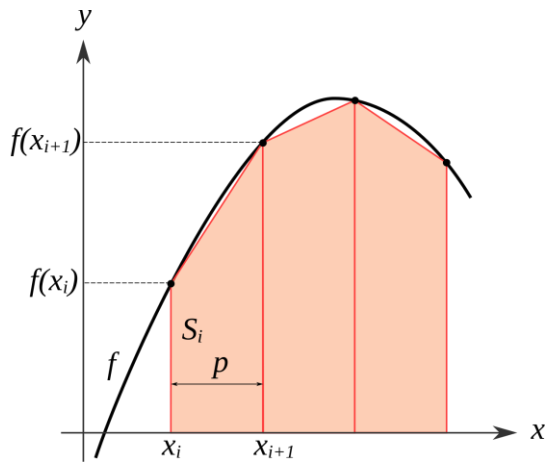
구와 광선의 충돌점을 구하는 방식으로 계산할 수 있다.

```
float dout = -r * mu + sqrt( r * r * ( mu * mu - 1.f ) + RL * RL );
```

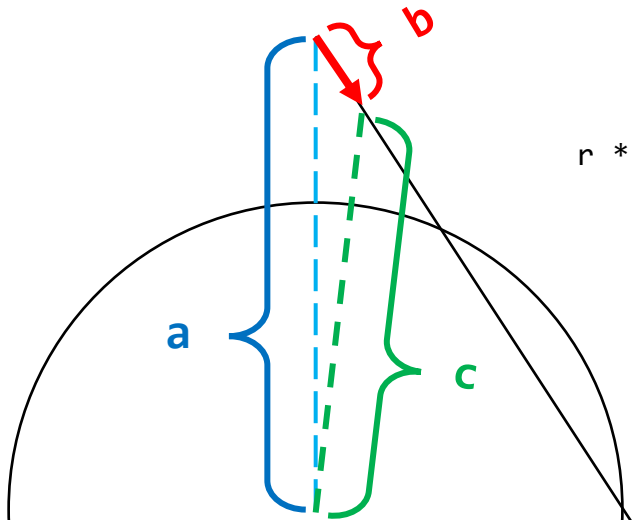


Transmittance Table

진행거리를 일정 간격으로 샘플링해서 $e^{\rho(h)X}$ 를 계산한 다음 사다리꼴 구분 구적법을 통해 특정 구간에 대한 광학적 깊이를 계산한다.



여기서 광선의 진행에 따라 변하는 고도를 제 2 코사인 법칙($c^2 = a^2 + b^2 - 2abcosC$)을 통해 재계산 한다.



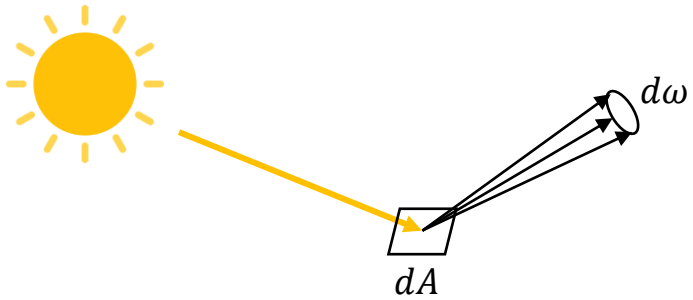

```
r * r + xj * xj + 2.f * xj * r * mu
```

Irradiance

태양 빛에 대한 복사 조도(Irradiance)가 다음으로 계산될 수치이다.

복사 조도는 단위 면적(dA)당 얼마나 많은 복사 속(Φ : $Radiant\ flux$) 을 받는지 나타낸 수치로 $E = \frac{\Phi}{dA}$ 로 표현할 수 있다.

이 단계에서 계산할 식을 설명하기 위해서는 복사 휘도(L : radiance)라는 개념을 알아야 하는데 복사 휘도는 복사 에너지가 어떤 방향으로 방출될 때, 그 방향으로 투영된 단위 면적으로부터, 단위 입체 각 당 방출되는 복사 속의 양이다.



복사 휘도는 $\mathrm{L} = \frac{\Phi}{d\omega dA cos\theta}$ 로 표현할 수 있고 복사 조도를 복사 휘도로 표현할 수 있다.

$$\mathrm{E} = \int_{\Omega} Lcos\theta d\omega$$

Single Irradiance Table

태양 빛에 대한 복사 조도는 태양을 점 광원으로 가정하고 대기에 의한 투과율을 반영하여 다음과 같이 간략하게 나타낼 수 있다.

$$\mathrm{E} = T(x, x_0) L_{sun} cos\theta$$

코드에서는 L_{sun}을 제외한 $\mathrm{E} = T(x, x_0)\, cos\theta$ 을 계산해서 저장한다.

```
[numthreads( IRRADIANCE_W / IRRADIANCE_GROUP_X, IRRADIANCE_H / IRRADIANCE_GROUP_Y, 1)]
void main( uint3 DTid : SV_DispatchThreadID )
{
	float r, muS;
	GetIrradianceRMuS( DTid, r, muS );
	irradiance1Buffer[DTid.xy] = float4( Transmittance( r, muS ) * max( muS, 0.f ), 0.f );
}
```

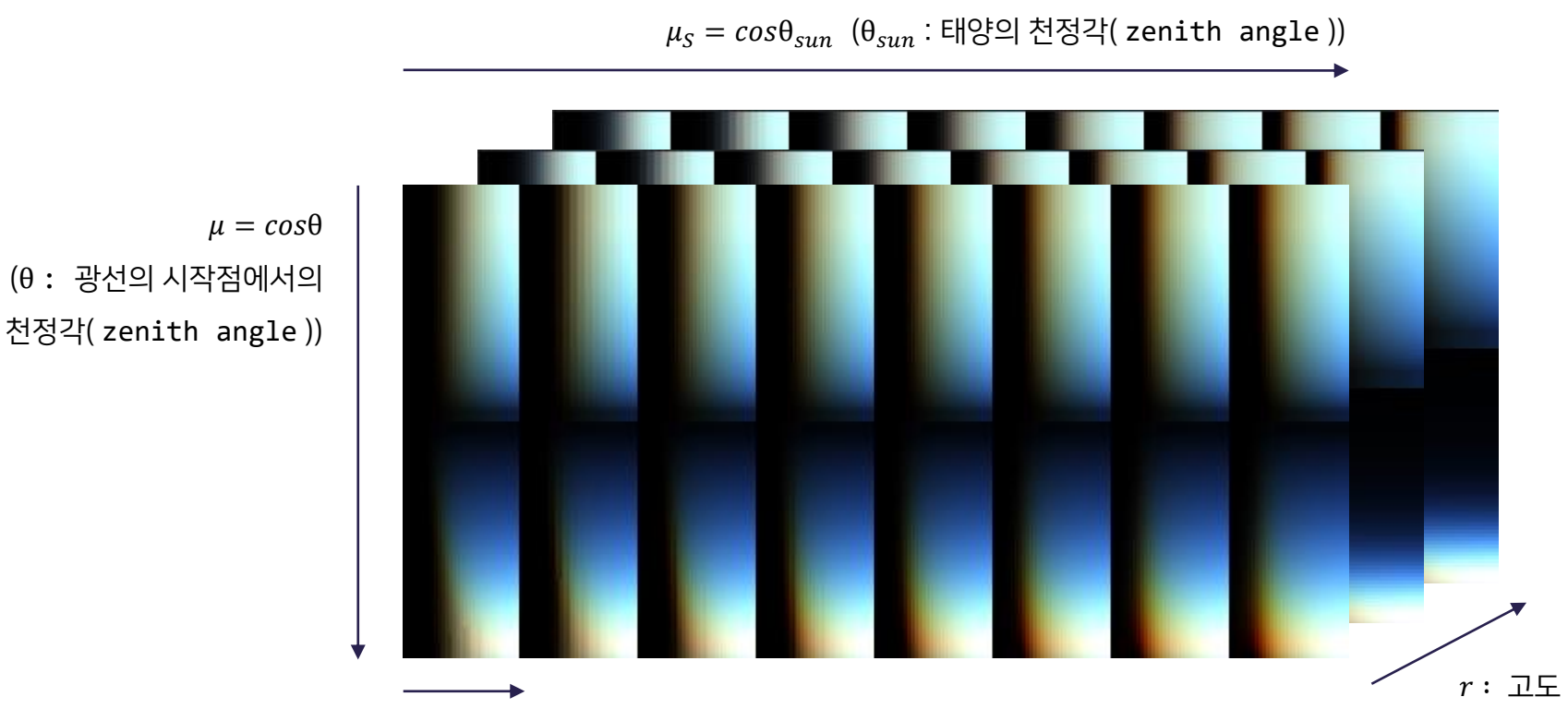
(θ : 광선의 시작점에의 천정각(zenith angle))

Single Inscatter Table

이제 태양의 대기 산란을 계산하도록 한다. 여기서는 레일리, 미의 산란 방정식을 따로 계산하여 3D 텍스처에 저장한다.

계산된 3D 텍스처는 4개의 인자로 LUT(Look Up Table)을 구성하고 있기 때문에 다음과 같다.

Rayleigh Inscatter Table의 일부 텍스처 모습



Phase Function

산란 방정식은 산란 계수 $\beta(\lambda, h)$를 이용해 다음과 같이 표기할 수 있다.

$$S(\lambda, \theta, h) = \beta(\lambda, h)\gamma(\theta)$$

$\gamma(\theta)$ 는 산란 방정식 $S(\lambda, \theta, h)$를 산란 계수 $\beta(\lambda, h)$ 로 나누는 것으로 구할 수 있다.

$$\gamma(\theta) = \frac{S(\lambda, \theta, h)}{\beta(\lambda, h)}$$

레일리 산란 방정식과 산란 계수를 통해 $\gamma(\theta)$을 구해보면

$$\gamma(\theta) = \overbrace{\frac{\pi^2(n^2-1)^2}{2}\frac{\rho(h)}{N}\frac{1}{\lambda^4}(1+\cos^2\theta)}^{S(\lambda,\theta,h)} \overbrace{\frac{3N\lambda^4}{8\pi^3(n^2-1)^2\rho(h)}}^{\frac{1}{\beta(\lambda,h)}}$$
$$= \frac{3}{16\pi}(1+\cos^2\theta)$$

파장에 상관없는 각도에 대한 함수라는 것을 알 수 있다.

$\gamma(\theta)$는 산란 각에 따른 빛의 세기를 나타내며 **위상 함수(Phase Function)**라 부른다.

Single Inscatter Table

레일리, 미 산란 방정식을 따로 저장하는 모습을 볼 수 있다.

```
[numthreads( RES_MU_S, RES_MU / INSCATTER1_GROUP_Y, 1)]
void main( uint3 DTid : SV_DispatchThreadID )
{
	float r;
	float4 dhdH;
	GetRdhdH( DTid.z, r, dhdH );

	float mu, muS, nu;
	GetMuMuSNu( DTid, r, dhdH, mu, muS, nu );

	float3 ray, mie;
	Inscatter( r, mu, muS, nu, ray, mie );

	deltaSRBuffer[DTid] = float4( ray, 0.f );
	deltaSMBuffer[DTid] = float4( mie, 0.f );
}
```

산란 방정식은 Inscatter 함수에서 계산한다.

```
void Inscatter( float r, float mu, float muS, float nu, out float3 ray, out float3 mie )
{
	ray = 0.f;
	mie = 0.f;
	float dx = limit( r, mu ) / float( INSCATTER_INTEGRAL_SAMPLES );
	float3 rayi;
	float3 miei;
	Integrand( r, mu, muS, nu, 0.f, rayi, miei );
	for ( int i = 1; i <= INSCATTER_INTEGRAL_SAMPLES; ++i )
	{
		float xj = float( i ) * dx;
		float3 rayj;
		float3 miej;
		Integrand( r, mu, muS, nu, xj, rayj, miej );
		ray += ( rayi + rayj ) * 0.5f * dx;
		mie += ( miei + miej ) * 0.5f * dx;
		rayi = rayj;
		miei = miej;
	}

	ray *= betaR;
	mie *= betaMSca;
}
```

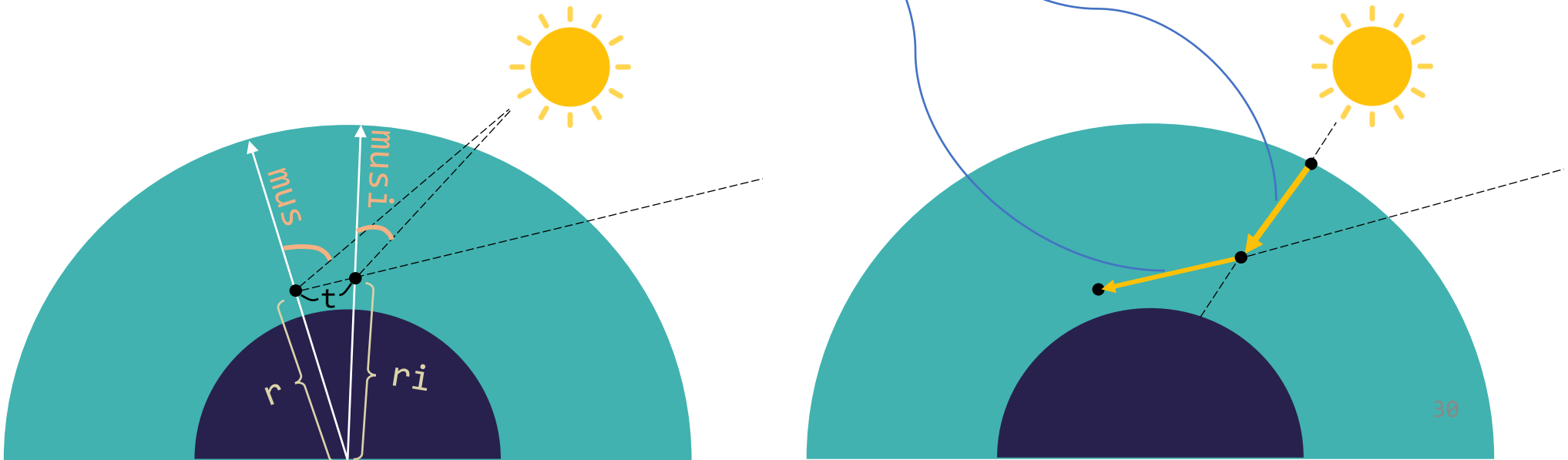
→ 산란 계수는 상수이므로 마지막에 적용한다.

※ 위상 함수는 정밀도의 문제로 실시간으로 계산하여 적용한다.

Single Inscatter Table

산란에 의해 빛의 방향이 변하기 때문에 변한 방향에 대한 투과율을 나눠서 적용해야 한다.

```
void Integrand( float r, float mu, float muS, float nu, float t, out float3 ray, out float3 mie )
{
	ray = 0.f;
	mie = 0.f;
	float ri = sqrt( r * r + t * t + 2.f * r * mu * t );
	float muSi = ( nu * t + muS * r ) / ri;
	ri = max( Rg, ri );

	// 지평선과 천정이 이루는 각도보다 태양과 천정이 이루는 각도가 작으면 계산
	// = 태양이 지구에 가려지면 산란 방정식의 값은 0
	if ( muSi >= -sqrt( 1.f - Rg * Rg / ( ri * ri ) ) )
	{
		float3 ti = Transmittance( r, mu, t ) * Transmittance( ri, muSi );
		ray = exp( -( ri - Rg ) / HR ) * ti;
		mie = exp( -( ri - Rg ) / HM ) * ti;
	}
}
```



Multiple Inscatter Table

지금까지 태양만을 고려한 복사 조도, 산란을 계산하였다. 이제는 지금까지 계산한 테이블을 바탕으로 복수의 산란현상을 시뮬레이션해야 한다.

최종 산란 텍스처에는 특이하게 값을 저장한다.

R	G	B	A
$C_* = S_R[L_{sun}] + \frac{S[L_*]}{P_R}$			$C_{M,r}$

$S_R[L_{sun}]$: 태양만을 고려한 레일리 산란 (위상 함수 적용 X)

$S[L_*]$: 다중 산란 (레일리, 미 포함 위상 함수 적용 ○)

P_R : 레일리 위상 함수

$C_{M,r}$: 미 산란의 red 채널 값 (위상 함수 적용 X)

```
[numthreads( RES_MU_S, RES_MU / INSCATTER1_GROUP_Y, 1)]
void main( uint3 DTid : SV_DispatchThreadID )
{
	float3 uvw = ( DTid.xyz + 0.5f ) / float3( RES_MU_S * RES_NU, RES_MU, RES_R );
	float4 ray = deltaSRTex.SampleLevel( deltaSRSampler, uvw, 0 );
	float4 mie = deltaSMTex.SampleLevel( deltaSMSampler, uvw, 0 );

	float pitch = RES_MU_S * RES_NU;
	float slicePitch = pitch * RES_MU;

	inscatterBuffer[( DTid.z * slicePitch ) + ( DTid.y * pitch ) + DTid.x] = float4( ray.rgb, mie.r );
}
```

$S_R[L_{sun}]$과 $C_{M,r}$로 산란 텍스처를 초기화 해준다. $\frac{S[L_*]}{P_R}$는 이 후에 계산되어 누적할 것이다.

Multiple Inscatter Table

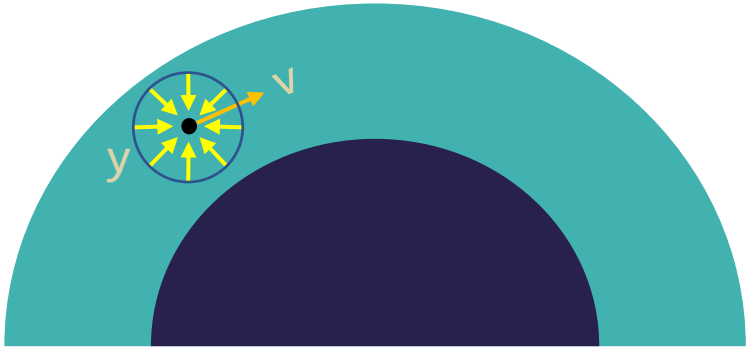
이렇게 저장하는 까닭은 LUT의 크기 제한으로 시선 벡터의 각도 정밀도가 제한될 수 밖에 없는데 미 산란이 각도에 강하게 영향을 받기 때문이다.

따라서 태양에 대한 미 산란의 **red 채널 값을 분리하여 저장**하고 실제 렌더링 시에 다음과 같이 복원한다.

$$C_M \simeq C_* \frac{C_{M,r}}{C_{*,r}} \frac{\beta_{R,r}^s}{\beta_{M,r}^s} \frac{\beta_M^s}{\beta_R^s}$$

다중 산란은 두 단계에 걸쳐서 계산되는데 우선 특정 위치 y에서 특정 방향 v로 산란되는 빛의 양을 계산한다.

```
[numthreads( RES_MU_S * RES_NU / INSCATTERS_GROUP_X, RES_MU / INSCATTERS_GROUP_Y, 1 )]
void main( uint3 DTid : SV_DispatchThreadID )
{
	float r;
	float4 dhdH;
	GetRdhdH( g_threadGroupZ, r, dhdH );

	float mu, muS, nu;
	GetMuMuSNu( float3( DTid.xy, g_threadGroupZ ), r, dhdH, mu, muS, nu );

	float3 raymie;
	Inscatter( r, mu, muS, nu, raymie );

	deltaJBuffer[DTid] = float4( raymie, 0.f );
}
```



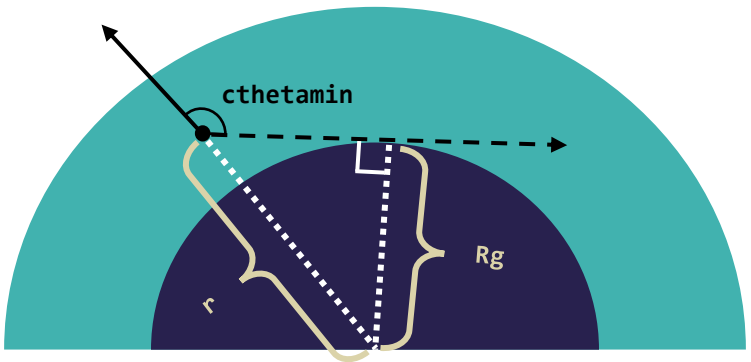
Multiple Inscatter Table

특정 위치 y로 들어오는 빛의 양을 계산하기 위하여 Inscatter 함수는 구 공간에 대한 적분을 계산하게 된다.

함수가 길기 때문에 여러 조각으로 쪼개서 살펴보자. 우선 수평선에 대한 천정각을 구한다.

이는 해당 빛이 지면에 반사되어 들어오게 된 빛인지 판단하기 위함이다.

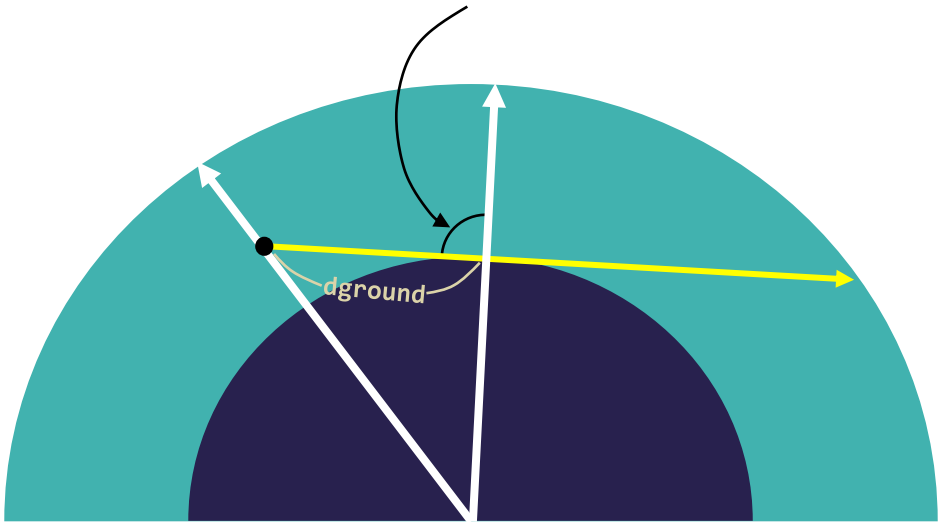
```
void Inscatter( float r, float mu, float muS, float nu, out float3 raymie )
{
	r = clamp( r, Rg, Rt );
	mu = clamp( mu, -1.f, 1.f );
	muS = clamp( muS, -1.f, 1.f );
	// Using formula sum and differences of cosine
	float var = sqrt( 1.f - mu * mu ) * sqrt( 1.f - muS * muS );
	nu = clamp( nu, mu * muS - var, mu * muS + var );

	// cos( theta to the horizon )
	float cthetamin = -sqrt( 1.f - ( Rg * Rg ) / ( r * r ) );

	float3 v = float3( sqrt( 1.0 - mu * mu ), 0.f, mu );
	float sx = v.x == 0.f ? 0.f : ( nu - muS * mu ) / v.x;
	float3 s = float3( sx, sqrt( max( 0.f, 1.f - sx * sx - muS * muS ) ), muS );

	raymie = 0.f;
```



Multiple Inscatter Table

수평선에 대한 천정각보다 특정 방향 w의 천정각이 더 크다면 해당 방향은 지면에서 반사된 빛이 들어오는 것이므로 지면의 반사로 인한 디퓨즈 항(greflectance)과 투과율(gtransp)을 계산해 준다.

```
// Compute equation(7)
// θ(0 ~ π)에 대한 적분
for ( int itheta = 0; itheta < INSCATTER_SPHERICAL_INTEGRAL_SAMPLES; ++itheta )
{
	float theta = ( float( itheta ) + 0.5f ) * dtheta;
	float ctheta = cos( theta );

	float greflectance = 0.f;
	float dground = 0.f;
	float3 gtransp = 0.f;

	// if ground visible in direction w compute transparency gtransp between x and ground
	if ( ctheta < cthetamin )
	{
		greflectance = AVERAGE_GROUND_REFLECTANCE / M_PI;
		dground = -r * ctheta - sqrt( r * r * ( ctheta * ctheta - 1.f ) + Rg * Rg );
		gtransp = Transmittance( Rg, -(r * ctheta + dground) / Rg , dground );
	}
```



Multiple Inscatter Table

지면에서 반사된 빛에 대한 렌더링 방정식을 계산한다.

$$L_o = L_e + \int_\Omega f(\omega_i, \omega_o) L_i (\omega \cdot n) d\omega$$

실제 계산해야 할 식은 다음과 같이 매우 간략화 할 수 있다.

$$L_o = \int_\Omega f(\omega_i, \omega_o) L_i (\omega \cdot n) d\omega = f(\omega_i, \omega_o) \int_\Omega L_i (\omega \cdot n) d\omega$$

지면에서 방출되는 빛 없음.

Lambertian BRDF 이므로 적분 밖으로 뺄 수 있음.

$$L_o = f(\omega_i, \omega_o) E$$

여기에 투과율을 적용해 주면 된다.

```
// ϕ(0 ~ 2π)에 대한 적분
for ( int iphi = 0; iphi < 2 * INSCATTER_SPHERICAL_INTEGRAL_SAMPLES; ++iphi )
{
	float phi = ( float( iphi ) + 0.5f ) * dphi;
	float dw = dtheta * dphi * sin( theta );
	float3 w = float3( cos( phi ) * sin( theta ), sin( phi ) * sin( theta ), ctheta );

	float nu1 = dot( s, w );
	float nu2 = dot( v, w );
	float pr2 = PhaseFunctionR( nu2 );
	float pm2 = PhaseFunctionM( nu2 );

	float3 gnormal = ( float3( 0.f, 0.f, r ) + dground * w ) / Rg;
	float3 girradiance = Irradiance( Rg, dot( gnormal, s ) );

	// first term = light reflected from the ground and attenuated before reaching x
	float3 raymie1 = greflectance * girradiance * gtransp;
```

Multiple Inscatter Table

태양에서의 빛이 두 번 꺾여야 하므로 (1. 태양의 입사 방향에서 특정 위치 y를 향하도록 2. 특정 위치 y에서 특정 방향 v를 향하도록) **산란 방정식은 2번 계산**된다.

다중 산란을 계산하는 과정은 이전 산란의 **누적**이다. 따라서 이 과정은 정해진 횟수 만큼 반복된다.

g_order는 계산해야 하는 다중 산란은 산란 횟수를 나타내며 처음으로 다중 산란을 계산하는 경우를 위한 특별한 처리를 위해 사용된다.

1

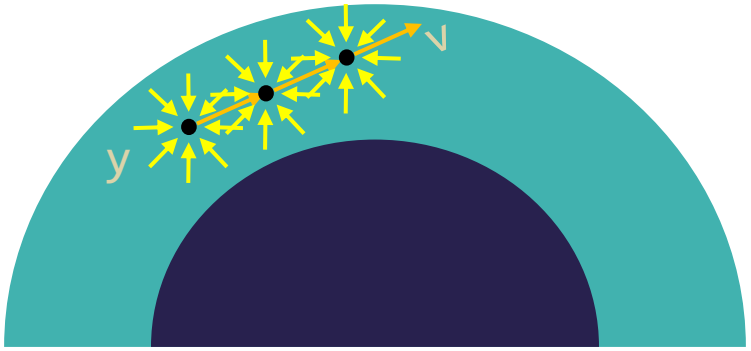
```
// second term = inscattered light = deltaS
// 태양으로 부터의 직접 산란을 계산한 경우를 1이라고 정했기 때문에 다중 산란을 처음 계산할 때는 2가 된다.
if ( g_order == 2 )
{
    // 단일 산란 계산 단계에서 미리 계산한 텍스처를 사용한다. 저장할 때 위상 함수를 적용하지 않았으므로 여기서 계산하여 적용해준다.
    float pr1 = PhaseFunctionR( nu1 );
    float pm1 = PhaseFunctionM( nu1 );
    float3 ray1 = Sample4D( deltaSRTex, deltaSRSampler, r, w.z, muS, nu1 ).rgb;
    float3 mie1 = Sample4D( deltaSMTex, deltaSMSampler, r, w.z, muS, nu1 ).rgb;

    raymie1 += ray1 * pr1 + mie1 * pm1;
}
else
{
    // 이전 단계의 다중 산란 계산의 결과 값에서 값을 가져온다.
    raymie1 += Sample4D( deltaSRTex, deltaSRSampler, r, w.z, muS, nu1 ).rgb;
}
```

2

```
// light coming from direction w and scattered in direction v
// Equation 7
raymie += raymie1 * ( betaR * exp( -( r - Rg ) / HR ) * pr2 + betaMSca * exp( -( r - Rg ) / HM ) * pm2 ) * dw;
    }
  }
}
```

Multiple Inscatter Table

다중 산란 계산을 위한 두번째 단계는 전 단계에서 계산한 값을 가지고 특정 진행 방향에 대한 산란을 계산하는 것이다.



특정 진행 방향에 대해서 일정 간격으로 샘플링하여 적분을 계산하게 된다.

```
[numthreads( RES_MU_S, RES_MU / INSCATTERN_GROUP_Y, 1 )]
void main( uint3 DTid : SV_DispatchThreadID )
{
	float r;
	float4 dhdH;
	GetRdhdH( DTid.z, r, dhdH );

	float mu, muS, nu;
	GetMuMuSNu( DTid, r, dhdH, mu, muS, nu );

	deltaSRBuffer[DTid] = float4( Inscatter( r, mu, muS, nu ), 0.f );
}
```

Multiple Inscatter Table

```
float3 Integrand( float r, float mu, float muS, float nu, float t )
{
	float ri = sqrt( r * r + t * t + 2.f * r * t * mu );
	float mui = ( r * mu + t ) / ri;
	float muSi = ( nu * t + muS * r ) / ri;
	return Sample4D( deltaJTex, deltaJSampler, ri, mui, muSi, nu ).rgb * Transmittance( r, mu, t );
}

float3 Inscatter( float r, float mu, float muS, float nu )
{
	float3 raymie = 0.f;
	float dx = limit( r, mu ) / float( INSCATTER_INTEGRAL_SAMPLES );
	float xi = 0.f;
	float3 raymiei = Integrand( r, mu, muS, nu, 0.f );
	for ( int i = 0; i < INSCATTER_INTEGRAL_SAMPLES; ++i )
	{
		float xj = float( i ) * dx;
		float3 raymiej = Integrand( r, mu, muS, nu, xj );
		raymie += ( raymiei + raymiej ) * 0.5f * dx;
		xi = xj;
		raymiei = raymiej;
	}

	return raymie;
}
```

적분은 투과율을 계산했던 것과 동일하게 사다리꼴 구분 구적법을 이용하여 계산한다.

Multiple Inscatter Table

최종 산란 텍스처에는 31번 슬라이드에서 언급했던 것 처럼 **레일리 위상 함수로 나눈 값**을 기록한다.

```
[numthreads( RES_MU_S, RES_MU / INSCATTERN_GROUP_Y, 1 )]
void main( uint3 DTid : SV_DispatchThreadID )
{
	float r;
	float4 dhdH;
	GetRdhdH( DTid.z, r, dhdH );

	float mu, muS, nu;
	GetMuMuSNu( DTid, r, dhdH, mu, muS, nu );

	float3 uvw = ( DTid.xyz + 0.5f ) / float3( RES_MU_S * RES_NU, RES_MU, RES_R );
	float pitch = RES_MU_S * RES_NU;
	float slicePitch = pitch * RES_MU;

	inscatterBuffer[( DTid.z * slicePitch ) + ( DTid.y * pitch ) + DTid.x] +=
	float4( deltaSRTex.SampleLevel( deltaSRSampler, uvw, 0 ).rgb / PhaseFunctionR( nu ), 0.f );
}
```

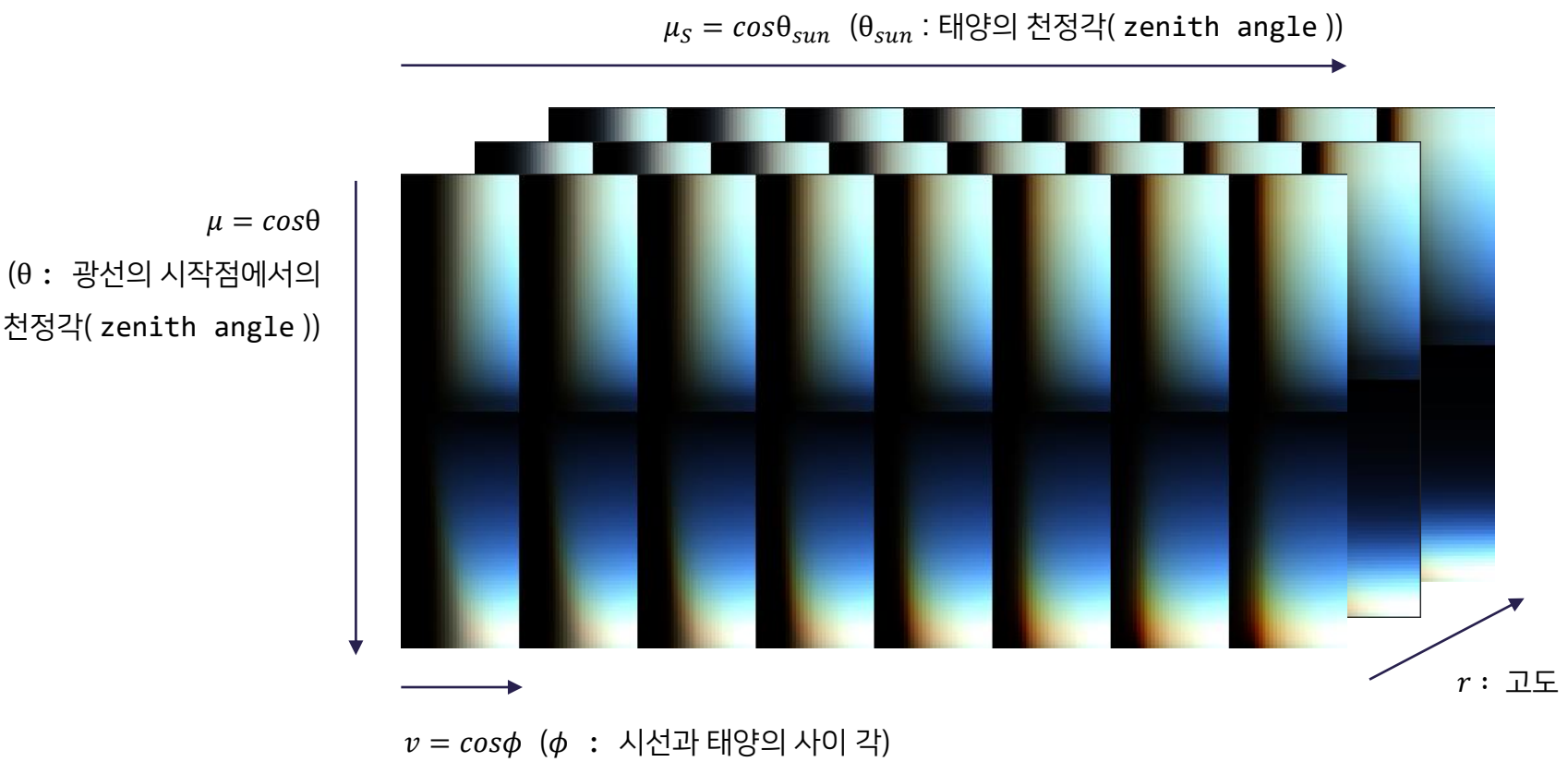
여기까지 계산하면 2번 산란이 일어났을 때 다중 산란을 계산한 것이 된다. 좀 더 현실적인 결과를 얻기 위해서 이 과정을 **일정 횟수 반복**한다.

Multiple Inscatter Table

```
// 4번의 다중 산란을 시뮬레이션
for ( int order = 2; order <= 4; ++order )
{
	int* gpuScatteringOrder = static_cast<int*>( renderer.LockBuffer( precomputeBuf ) );
	gpuScatteringOrder[0] = order;
	renderer.UnLockBuffer( precomputeBuf );

	renderer.BindConstantBuffer( SHADER_TYPE::CS, 0, 1, &precomputeBuf );

	// Compute deltaJ
	renderer.BindShader( SHADER_TYPE::CS, csInscatterS );
	renderer.BindShaderResource( SHADER_TYPE::CS, 1, 1, &deltaESrv );
	renderer.BindRandomAccessResource( 0, 1, &deltaJRav );

	// 한번에 하려고 하니 처리시간이 오래걸려 Device Lost가 발생. 적당히 나눠서 Dispatch.
	for ( int i = 0; i < INSCATTERS_GROUP_Z; ++i )
	{
		gpuScatteringOrder = static_cast<int*>( renderer.LockBuffer( precomputeBuf ) );
		gpuScatteringOrder[0] = order;
		gpuScatteringOrder[1] = i;
		renderer.UnLockBuffer( precomputeBuf );

		renderer.BindConstantBuffer( SHADER_TYPE::CS, 0, 1, &precomputeBuf );

		renderer.Dispatch( INSCATTERS_GROUP_X, INSCATTERS_GROUP_Y );
		renderer.WaitGPU( );
	}
	…

	// Compute deltaS
	renderer.BindShader( SHADER_TYPE::CS, csInscatterN );
	renderer.BindShaderResource( SHADER_TYPE::CS, 4, 1, &deltaJSrv );
	renderer.BindShaderResource( SHADER_TYPE::CS, 2, 1, nullptr );
	renderer.BindRandomAccessResource( 0, 1, &deltaSRRav );

	renderer.Dispatch( INSCATTERN_GROUP_X, INSCATTERN_GROUP_Y, INSCATTERN_GROUP_Z );
	…

}
```

Multiple Inscatter Table

최종 Inscatter Table



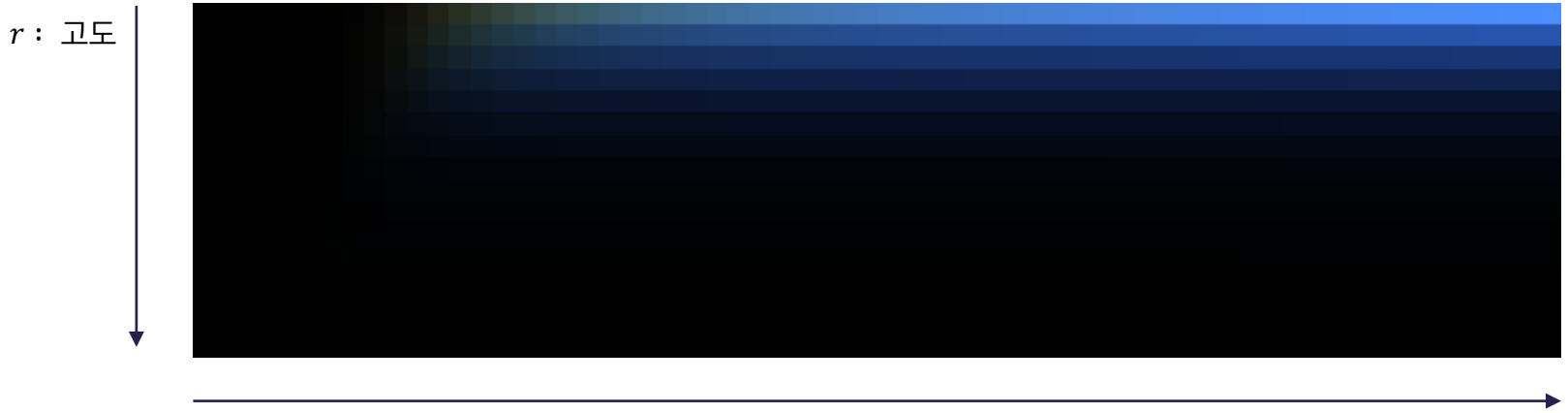
Multiple Irrdiance Table

복사조도도 다중 산란에 의한 빛의 기여를 고려해야 한다.

```
[numthreads( IRRADIANCE_W / IRRADIANCE_GROUP_X, IRRADIANCE_H / IRRADIANCE_GROUP_Y, 1 )]
void main( uint3 DTid : SV_DispatchThreadID )
{
	float r, muS;
	GetIrradianceRMuS( DTid, r, muS );

	float3 s = float3( max( sqrt( 1.0 - muS * muS ), 0.f ), 0.f, muS );

	float3 result = 0.f;
	// 반구영역에 대해서 적분
	for ( int iphi = 0; iphi < 2 * IRRADIANCE_INTEGRAL_SAMPLES; ++iphi )
	{
		float phi = ( float( iphi ) + 0.5f ) * dphi;
		for ( int itheta = 0; itheta < IRRADIANCE_INTEGRAL_SAMPLES / 2; ++itheta )
		{
			float theta = ( float( itheta ) + 0.5f ) * dtheta;
			float dw = sin( theta ) * dtheta * dphi;
			float3 w = float3( cos( phi ) * sin( theta ), sin( phi ) * sin( theta ), cos( theta ) );
			float nu = dot( s, w );
			// 다중 산란 계산과 마찬가지로 첫단계에서는 위상 함수를 실시간으로 계산하여 적용해 준다.
			if ( g_order == 2 )
			{
				float pr = PhaseFunctionR( nu );
				float pm = PhaseFunctionM( nu );
				float3 ray = Sample4D( deltaSRTex, deltaSRSampler, r, w.z, muS, nu ).rgb;
				float3 mie = Sample4D( deltaSMTex, deltaSMSampler, r, w.z, muS, nu ).rgb;
				result += ( ray * pr + mie * pm ) * w.z * dw;
			}
			else
			{
				result += Sample4D( deltaSRTex, deltaSRSampler, r, w.z, muS, nu ).rgb * w.z * dw;
			}
		}
	}

	irradianceBuffer[DTid.xy] = float4( result, 0.f );
}
```

Multiple Irradiance Table

최종 Irradiance Table



(θ : 광선의 시작점에의 천정각(zenith angle))

Rendering

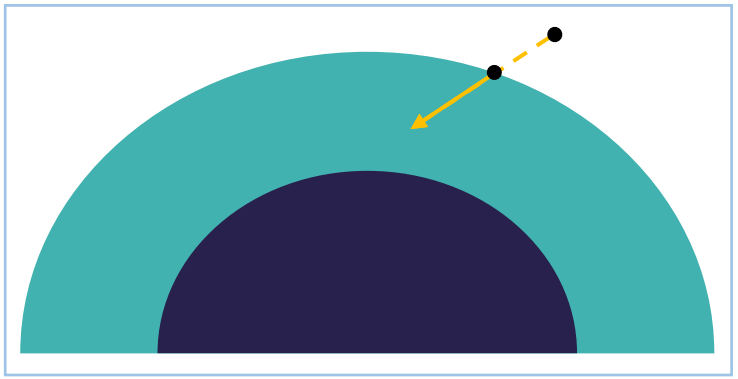
```
VS_OUTPUT main( uint vertexID : SV_VertexID )
{
	VS_OUTPUT output = (VS_OUTPUT)0;

	float2 uv = float2( ( vertexID & 1 ) << 1, vertexID & 2 );
	float4 pos = float4( float2( -1.f, 1.f ) + uv * float2( 2.f, -2.f ), 0.f, 1.f );

	output.position = pos;

	float4x4 invVewProjection = mul( g_invProjectionMatrix, g_invViewMatrix );
	float4 worldPos = mul( float4( pos.xy, 1.f, 1.f ), invVewProjection );
	output.worldPos = worldPos.xyz / worldPos.w;

	return output;
}
```

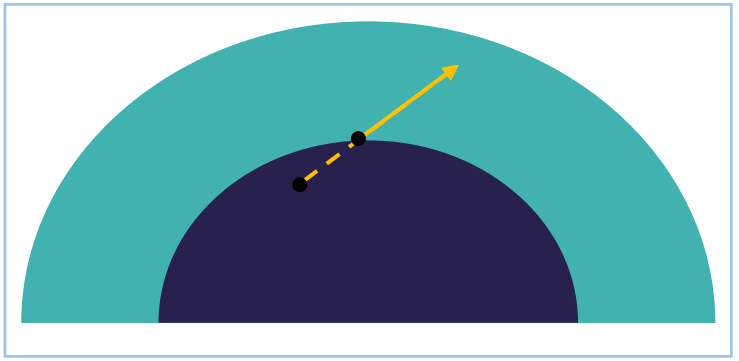
픽셀 셰이더는 산란에 의한 색, 땅의 색, 태양에 의한 색을 합하여 최종 색상을 계산한다.

```
float4 main( PS_INPUT input ) : SV_Target
{
	float3 viewRay = normalize( input.wolrdPos - g_cameraPos.xyz );

	float scale = 0.00001f; // 길이 단위를 맞추기 위한 스케일 값
	float3 viewPos = g_cameraPos.xyz * scale;
	viewPos.y += Rg + HeightOffset;

	float r = length( viewPos );

	float mu = dot( viewPos, viewRay ) / r;
	float t = -r * mu - sqrt( r * r * ( mu * mu - 1.f ) + Rg * Rg );

	float3 attenuation;
	float3 inscatterColor = InscatterColor( viewPos, t, viewRay, r, mu, attenuation );
	float3 groundColor = GroundColor( viewPos, t, viewRay, r, mu, attenuation );
	float3 sunColor = SunColor( viewPos, t, viewRay, r, mu );

	return float4( HDR( sunColor + inscatterColor + groundColor ), 1.f );
}
```

Rendering - Inscatter Color

```
float3 InscatterColor( float3 x, float t, float3 viewRay, float r, float mu, out float3 attenuation )
{
	float3 result = 0.f;
	attenuation = 1.f;
	// 관찰 위치가 대기 밖일 때 대기를 벗어나지 않도록 조정
	float d = -r * mu - sqrt( r * r * ( mu * mu - 1.f ) + Rt * Rt );
	if ( d > 0.f )
	{
		x += d * viewRay;
		t -= d;
		mu = ( r * mu + d ) / Rt;
		r = Rt;
	}

	static const float EPS = 0.005f;
	// 관찰 위치가 땅보다 내려가지 않도록 조정
	if ( r < Rg + HeightOffset + EPS )
	{
		float diff = ( Rg + HeightOffset + EPS ) - r;
		x -= diff * viewRay;
		t -= diff;
		mu = dot( x, viewRay ) / r;
		r = Rg + HeightOffset + EPS;
	}

	if ( r <= Rt )
	{
		float nu = dot( viewRay, g_sunDir.xyz );
		float muS = dot( x, g_sunDir.xyz ) / r;
		float4 inscatter = max( Sample4D( inscatterTex, inscatterSampler, r, mu, muS, nu ), 0.f );

		if ( t > 0.f )
		{
			float3 x0 = x + t * viewRay;
			float altitude0 = length( x0 );
			float mu0 = dot( x0, viewRay ) / altitude0;
			float muS0 = dot( x0, g_sunDir.xyz ) / altitude0;
			// 정밀도 문제를 피하기 위해서 투과율을 실시간으로 계산
			attenuation = AnalyticTransmittance( r, mu, t );
```



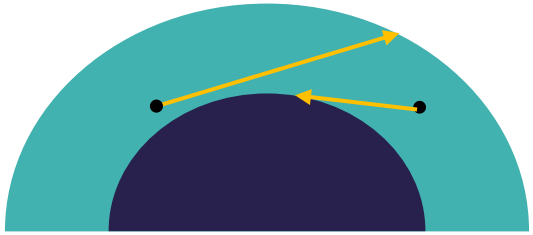


Rendering - Inscatter Color

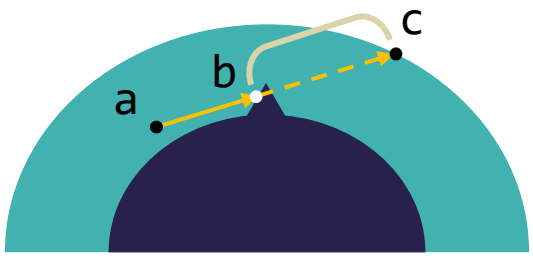
```
                if ( altitude0 > Rg + HeightOffset )
                {
                        // 지형에 가려진 구간의 산란은 제외
                        inscatter = max( inscatter - attenuation.rgbr * Sample4D( inscatterTex, inscatterSampler, altitude0,
        mu0, muS0, nu ), 0.f );

                        // 지평선에 가까울 때 발생하는 정밀도 문제를 회피하기 위해서 지평선의 위 아래의 산란 수치를 보간
                        float muHoriz = -sqrt( 1.f - ( Rg / r ) * ( Rg / r ) );
                        if ( abs( mu - muHoriz ) < EPS )
                        {
                                float a = ( ( mu - muHoriz ) + EPS ) / ( 2.f * EPS );

                                mu = muHoriz - EPS;
                                altitude0 = sqrt( r * r + t * t + 2.f * r * t * mu );
                                mu0 = ( r * mu + t ) / altitude0;

                                float4 inscatter0 = Sample4D( inscatterTex, inscatterSampler, r, mu, muS, nu );
                                float4 inscatter1 = Sample4D( inscatterTex, inscatterSampler, altitude0, mu0, muS0, nu );
                                float4 inscatterA = max( inscatter0 - attenuation.rgbr * inscatter1, 0.f );

                                mu = muHoriz + EPS;
                                altitude0 = sqrt( r * r + t * t + 2.f * r * t * mu );
                                mu0 = ( r * mu + t ) / altitude0;

                                inscatter0 = Sample4D( inscatterTex, inscatterSampler, r, mu, muS, nu );
                                inscatter1 = Sample4D( inscatterTex, inscatterSampler, altitude0, mu0, muS0, nu );
                                float4 inscatterB = max( inscatter0 - attenuation.rgbr * inscatter1, 0.f );

                                inscatter = lerp( inscatterA, inscatterB, a );
                        }
                }
        }

        inscatter.w *= smoothstep( 0.f, 0.02f, muS );

        float phaseR = PhaseFunctionR( nu );
        float phaseM = PhaseFunctionM( nu );
        // Mie 산란을 복원
        result = max( inscatter.rgb * phaseR + getMie( inscatter ) * phaseM, 0.f );
    }

    return result * ISun;
}
```

Rendering - Inscatter Color

Transmittance Table과 마찬가지로 Inscatter Table은 어떤 위치에서 땅이나 대기 최상단에 닿을 때 까지 특정 방향으로 진행하는 빛의 양이 어느 정도되는 지를 계산하여 저장해 놓았다.



진행 경로상에 지형과 같은 방해물이 있을 경우에는 가려진 구간은 제외한 값을 구해야 한다.



위 그림의 a ~ c 구간 산란 방정식을 $S(a \rightarrow c) = \int_a^c TJ[\overline{L_0}]$라 하면 이를 a ~ b와 b ~ c 구간의 산란 방적식으로 분리하여 다음과 같이 표현할 수 있다.

$$\int_a^c TJ[\overline{L_0}] = \int_a^b TJ[\overline{L_0}] + \int_b^c TJ[\overline{L_0}]$$

Rendering - Inscatter Color

이를 a ~ b 구간에 대해서 정리하면

$$\int_a^b TJ[\overline{L_0}] = \int_a^c TJ[\overline{L_0}] - \int_b^c TJ[\overline{L_0}]$$

가 되고 최종적으로는 다음과 같이 정리할 수 있다.

$$\mathrm{S}(\mathrm{a} \to b) = \mathrm{S}(\mathrm{a} \to \mathrm{c}) - T(a \to b)\mathrm{S}(\mathrm{b} \to \mathrm{c}) \quad \because T = T(a \to c) = T(a \to b)T(b \to c)$$

코드에서는 다음과 같다.

```
inscatter = max( inscatter - attenuation.rgbr * Sample4D( inscatterTex, inscatterSampler, altitude0, mu0, muS0, nu ), 0.f );
float4 inscatterA = max( inscatter0 - attenuation.rgbr * inscatter1, 0.f );
float4 inscatterB = max( inscatter0 - attenuation.rgbr * inscatter1, 0.f );
```

32번 ppt에서 이야기한 Mie 산란의 복원은 getMie 함수에서 수행한다.
getMie 함수는 $C_M \simeq C_* \frac{C_{M,r}}{C_{*,r}} \frac{\beta_{R,r}^s}{\beta_{M,r}^s} \frac{\beta_M^s}{\beta_R^s}$를 다음과 같이 간단히 정리하여 작성되었다.

$$C_M \simeq C_* \frac{C_{M,r}}{C_{*,r}} \frac{\beta_{R,r}^s}{\beta_R^s} (\because \beta_{M,r}^s = \beta_{M,g}^s = \beta_{M,b}^s)$$

```
return rayMie.rgb * rayMie.w / max( rayMie.r, 1e-4 ) * ( betaR.r / betaR );
```

Rendering – Ground Color

땅의 색은 디퓨즈 색상을 계산한다. 입사광으로써 산란으로 인한 빛과 태양 빛이 고려된다.

```
float3 GroundColor( float3 x, float t, float3 viewRay, float r, float mu, float3 attenuation )
{
	float3 result = 0.f;

	if ( t > 0.f )
	{
		float3 x0 = x + t * viewRay;
		float altitude0 = length( x0 );
		float3 n = x0 / altitude0;

		float muS = dot( n, g_sunDir.xyz );
		// 지평선에 가려지는 경우는 투과율이 0이다.
		float3 sunLight = TransmittanceWithShadow( altitude0, muS );

		float3 groundSkyLight = Irradiance( altitude0, muS );

		float3 sceneColor = float3( 0.35f, 0.35f, 0.35f );
		float3 reflectance = sceneColor * float3( 0.2f, 0.2f, 0.2f );
		if ( altitude0 > Rg + HeightOffset )
		{
			reflectance = float3( 0.4f, 0.4f, 0.4f );
		}

		 // 렌더링 방정식 계산
		float3 groundColor = reflectance * ISun / M_PI * ( max( muS, 0.f ) ) * ( sunLight + groundSkyLight );

		result = attenuation * groundColor;
	}

	return result;
}
```

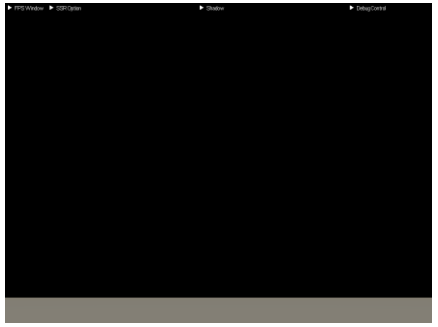
Rendering – Sun Color

태양의 색은 시야 벡터와 태양의 방향 벡터가 이루는 각이 1도 보다 작을 때 태양의 복사 세기(Radiant Intensity)에 투과율을 적용시켜 반환한다.

```
float3 SunColor( float3 x, float t, float3 viewRay, float r, float mu )
{
	if ( t > 0.f )
	{
		return 0.f;
	}
	else
	{
		float3 transmittance = r <= Rt ? TransmittanceWithShadow( r, mu ) : 1.f;
		float isun = step( cos( M_PI / 180 ), dot( viewRay, g_sunDir.xyz ) ) * ISun;
		return transmittance * isun;
	}
}
```

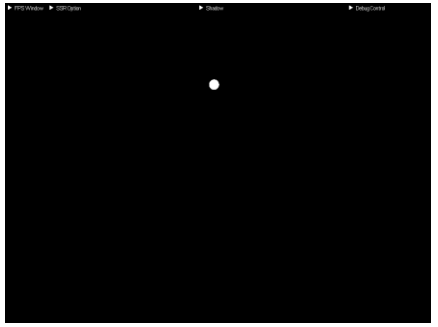
Inscatter color

+



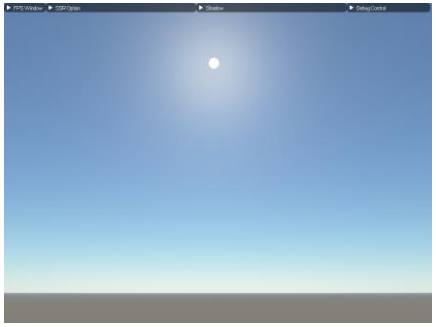
Ground color

+



Sun color

=



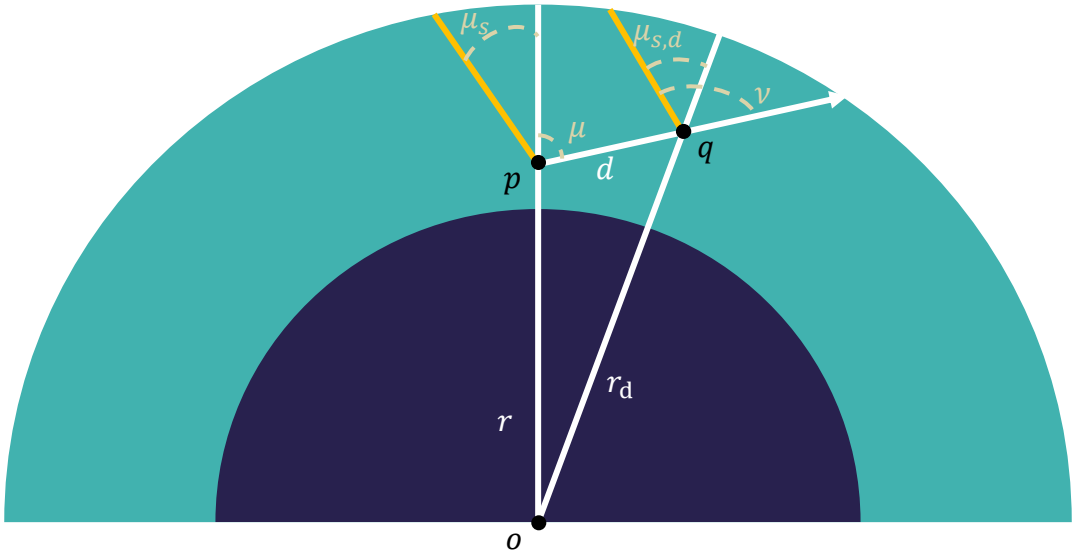
Final result

Reference

- 아이콘 이미지 : https://www.flaticon.com/authors/simpleicon
- https://www.alanzucconi.com/2017/10/10/atmospheric-scattering-1/
- https://ebruneton.github.io/precomputed_atmospheric_scattering/

Appendix

특정 위치에서 태양에 대한 천정각 구하기



$$r = ||op||, d = ||pq||$$

$$\mu = \frac{op \cdot pq}{rd}, \mu_s = \frac{op \cdot \omega_s}{r}, \nu = \frac{pq \cdot \omega_s}{d}$$

$$r_d = ||oq|| = \sqrt{d^2 + 2r\mu d + r^2}$$

$$\mu_{s,d} = \frac{oq \cdot \omega_s}{r_d} = \frac{(op + pq) \cdot \omega_s}{r_d} = \frac{r\mu_s + d\nu}{r_d}$$

```
float muSi = ( nu * t + muS * r ) / ri; // 38번 ppt
```