Outdoor Light Scattering Sample Update

Egor Yusov

egor.a.yusov@intel.com

Introduction

This document describes an update to the previously published Outdoor Light Scattering sample[1]. The following new features were implemented:

- Multiple scattering of sun light in the atmosphere.
    - Multiple scattering has subtle effect during the day time, but makes the sky look much more natural.
    - During the twilight, multiple scattering has higher importance add improves realism significantly (Figure 1).
    - Light radiance due to multiple scattering is pre-computed for all possible locations and directions during initialization and stored in a 4-dimensional look-up table.
- Faster method for rendering light shafts using pre-computed look-up table.
    - Since light radiance is pre-computed, this new method do not perform numerical integration of air light integral, but only computes total length of the illuminated ray fraction and distance to the first lit point. Two look-ups into the scattering look-up table are then performed.
    - This method is faster than numerical integration, but causes some artifacts due to significant non-linearity of the data stored in the look-up table.
- Physically-based shaders for rendering Earth surface using correct sun light extinction and pre-computed ambient skylight look-up table.
    - The sun light extinction due to the atmosphere is evaluated for each vertex of the Earth mesh; pre-computed ambient skylight texture is used to estimate ambient light.
- Various tone mapping operators.
    - The sample implements a number of different tone mapping operators that perform HDR to LDR conversion.

[1] http://software.intel.com/en-us/blogs/2013/06/26/outdoor-light-scattering-sample

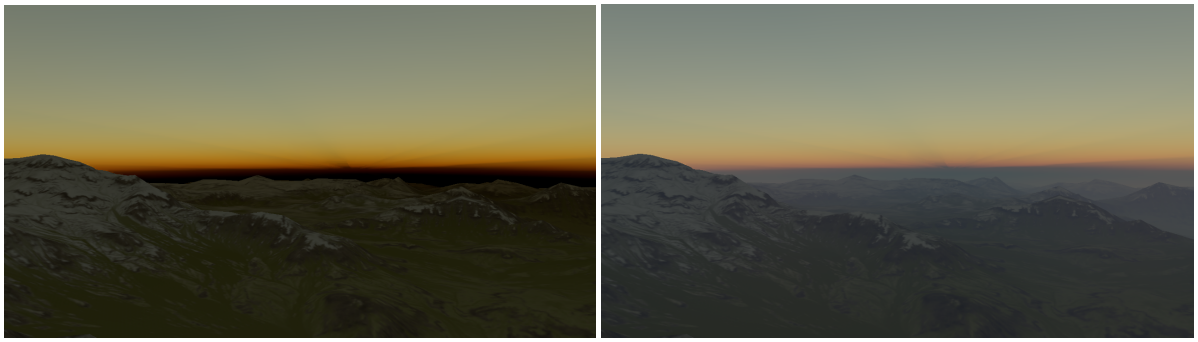


Figure 1. Twilight sky rendered with single scattering (left) and multiple scattering (right).

Multiple Scattering

Multiple scattering is the most important new feature implemented in the sample update. The implementation follows the method described in [Bruneton and Neyret 08]. The main idea of this method is that the light radiance due to multiple scattering $L_{In}^{M}(C,\vec{v},\vec{l})$ at point C when looking in direction $\vec{v}$ and sun is in direction $\vec{l}$ can be pre-computed and stored in a 4-dimensional look-up table. In our implementation, we use simplified pre-computation algorithm which ignores Earth surface reflection. It can be summarized as follows:

1. Compute single scattering $L_{In}^{(1)}(C,\vec{v},\vec{l})$ with trapezoidal integration, for all possible $C,\vec{v},\vec{l}$.

2. For every higher order scattering *n* from 2 to N:

   a. Compute order *n* scattered light radiance $J^{(n)}(C,\vec{v},\vec{l})$ at every point of space C and all directions $\vec{v}$ and $\vec{l}$ by integrating previous order scattering $L_{In}^{(n-1)}$ over the whole set of directions $\vec{\omega}$:

   $$J^{(n)}(C,\vec{v},\vec{l}) = \int_{4\pi} \left(\beta_R^s e^{-h/H_R} p_R(\vec{\omega}\bullet\vec{l}) + \beta_M^s e^{-h/H_M} p_M(\vec{\omega}\bullet\vec{l})\right)\cdot L_{In}^{(n-1)}(C,\vec{\omega},\vec{l})\cdot d\omega, \qquad (1)$$

   where *h* is the height of point C above sea level.

   b. Compute order *n* in-scattering radiance for every point of space, view direction and light direction by integrating scattered radiance $J^{(n)}$ along the view ray:

   $$L_{In}^{(n)}(C,\vec{v},\vec{l}) = \int_{C}^{X_T} e^{-T(P(s)\to C)}\cdot J^{(n)}(P(s),\vec{v},\vec{l})\cdot ds, \qquad (2)$$

   where $P(s) = C + \vec{v}\cdot s$, and X_T is the point where the ray $C+\vec{v}\cdot s$ terminates, i.e. hits the top of the atmosphere or the Earth surface.

   c. Add order *n* in-scattering to the total higher order-scattering look-up table: $L_{In}^{H} = L_{In}^{H} + L_{In}^{(n)}$.

3. Compute multiple scattering look-up table as the sum of single and higher order scattering: $L_{In}^{M} = L_{In}^{H} + L_{In}^{(1)}$.

In our implementation we support three look-up tables:

- `tex3DSingleScattering` stores single scattering only ($L_{In}^{(1)}$);
- `tex3DHighOrderScattering` stores higher order scattering only (L_{In}^{H});
- `tex3DMultipleScattering` stores multiple scattering (L_{In}^{M}).

After multiple scattering look-up table L_{In}^{M} is ready, we compute one-dimensional ambient sky light look-up table according to the following equation:

$$E_{Sky}(\varphi) = \int_{2\pi} L_{In}^{M}(C, \vec{\omega}, \vec{l}(\varphi)) \cdot (\vec{\omega} \bullet \vec{N}_{Earth}) \cdot d\vec{\omega}, \tag{3}$$

where integration is performed over the upper hemisphere, and φ is the angle between light direction $\vec{l}$ and the Earth normal $\vec{N}_{Earth}$ (note that any vector $\vec{l}$ which makes angle φ with $\vec{N}_{Earth}$ can be used in the equation). The camera C is assumed to be located on the ground.

Look-up table parameterization

As shown by Bruneton and Neyret [Bruneton and Neyret 08], the look-up table can be parameterized by 4 variables: height of the point above sea level (altitude), and cosine of three angles: the angle between the view direction ($\vec{v}$) and zenith ($\vec{N}_{Earth}$), the direction on sun ($\vec{l}$) and zenith, and the angle between view and sun directions. We will denote these parameters by h, $c_{\alpha} = \cos\alpha = (\vec{v} \bullet \vec{N}_{Earth})$, $c_{\varphi} = \cos\varphi = (\vec{l} \bullet \vec{N}_{Earth})$ and $c_{\theta} = \cos\theta = (\vec{v} \bullet \vec{l})$. Unfortunately, simple linear scaling of these parameters results in severe artifacts near horizon (Figure 2, left). Bruneton and Neyret [Bruneton and Neyret 08] proposed a more efficient non-linear parameterization which eliminates most of the artifacts. We however found that some issues still remain when sun is close to horizon (Figure 2, right).

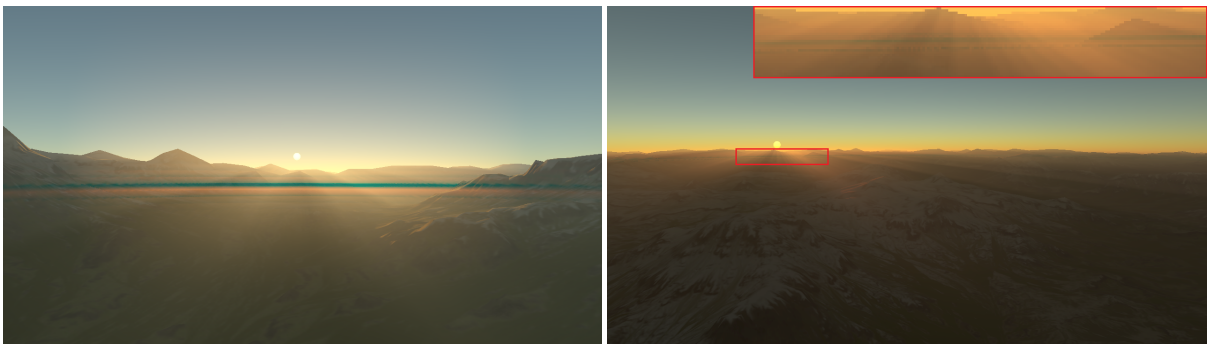


Figure 2. Artifacts near horizon when using linear parameterization (left) and Bruneton and Neyret's non-linear parameterization (right).

We thus use a different parameterization ($u_h, u_\alpha, u_\varphi, u_\theta$) described below:

$$u_h = \left(\frac{h}{H_{Atm}} \right)^{0.5}, \tag{4}$$

where $H_{Atm} = 80000$ is the height of the top of the atmosphere.

$$u_\alpha = \begin{cases} 0.5\left(\frac{c_\alpha - c_h}{1 - c_h} \right)^{0.2} + 0.5, \; c_\alpha > c_h \\ 0.5\left(\frac{c_h - c_\alpha}{c_h - (-1)} \right)^{0.2}, \; c_\alpha \le c_h \end{cases} \tag{5}$$

where $c_h = -\frac{\sqrt{h \cdot (2R_{Earth} + h)}}{R_{Earth} + h}$ is the cosine of the horizon angle for altitude *h*.

We use the same parameterization for c_φ as in Bruneton's implementation:

$$u_\varphi = 0.5\left(\arctan(\max(c_\varphi, -0.1975) \cdot \tan(1.26 \cdot 1.1)) / 1.1 + (1.0 - 0.26)\right). \tag{6}$$

Finally, parameterization for c_θ is as follows:

$$u_\theta = \begin{cases} 0.5 + 0.5 \cdot \left(2 \cdot (\gamma - 0.5)\right)^{1.5}, \; \gamma \ge 0.5 \\ 0.5 - 0.5 \cdot \left(2 \cdot (0.5 - \gamma)\right)^{1.5}, \; \gamma < 0.5 \end{cases} \tag{7}$$

where $\gamma = \frac{\arccos(c_\theta)}{\pi} = \frac{\theta}{\pi}$.

Figure 3 depicts the above non-linear parameterization (u_α is given for *h* = 20000).

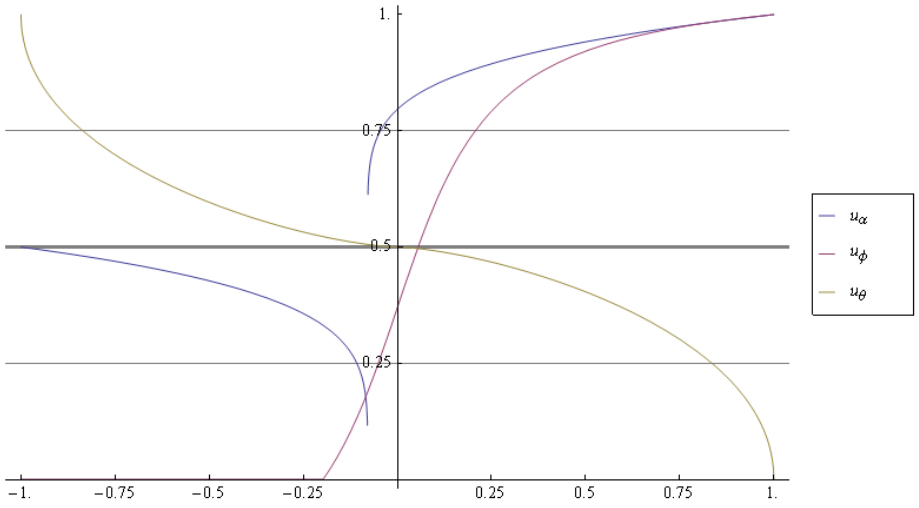


Figure 3. Non-linear parameterization.

This parameterization helps eliminate artifacts at the horizon, as shown in Figure 4.

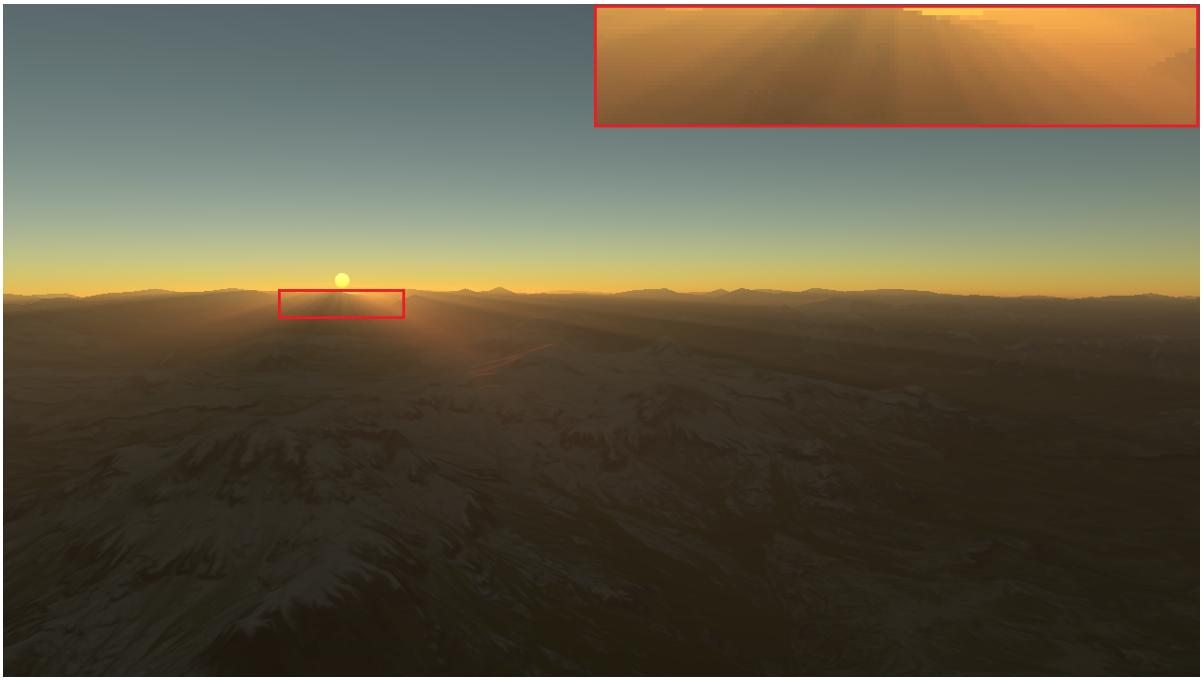


Figure 4. Our parameterization does not exhibit apparent artifacts.

It must be noted that due to substantial non-linearity of the radiance data stored in the look-up table, all the artifacts can be fixed only by impractical increase of the look-up table resolution. In our implementation, we use 32x128x64x16 16-bit float look-up tables requiring 32 MB of video memory. Lower resolution tables can be used to tread minor quality degradation for reduced memory consumption.

Performing Look-ups into the Table

Since 4D textures are not natively supported by the current graphics hardware, our 4D look-up table is packed into a 3D texture. Consequently, manual linear interpolation for the fourth coordinate is required. The source code of the function that performs one look-up into the provided table is given in Listing 1.

```
float3 LookUpPrecomputedScattering(float3 f3StartPoint,
                                   float3 f3ViewDir,
                                   float3 f3EarthCentre,
                                   float3 f3DirOnLight,
                                   in Texture3D<float3> tex3DScatteringLUT,
                                   inout float4 f4UVWQ)
{
    float3 f3EarthCentreToPointDir = f3StartPoint - f3EarthCentre;
    float fDistToEarthCentre = length(f3EarthCentreToPointDir);
    f3EarthCentreToPointDir /= fDistToEarthCentre;
    float fHeightAboveSurface = fDistToEarthCentre - EARTH_RADIUS;
    float fCosViewZenithAngle = dot( f3EarthCentreToPointDir, f3ViewDir    );
    float fCosSunZenithAngle  = dot( f3EarthCentreToPointDir, f3DirOnLight );
    float fCosSunViewAngle    = dot( f3ViewDir,               f3DirOnLight );

    // Provide previous look-up coordinates
    f4UVWQ = WorldParams2InsctrLUTCoords(fHeightAboveSurface, fCosViewZenithAngle,
                                         fCosSunZenithAngle, fCosSunViewAngle,
                                         f4UVWQ);

    float3 f3UVW0;
    f3UVW0.xy = f4UVWQ.xy;
    float fQ0Slice = floor(f4UVWQ.w * PRECOMPUTED_SCTR_LUT_DIM.w - 0.5);
    fQ0Slice = clamp(fQ0Slice, 0, PRECOMPUTED_SCTR_LUT_DIM.w-1);
    float fQWeight = (f4UVWQ.w * PRECOMPUTED_SCTR_LUT_DIM.w - 0.5) - fQ0Slice;
    fQWeight = max(fQWeight, 0);
    float2 f2SliceMinMaxZ =
        float2(fQ0Slice, fQ0Slice+1)/PRECOMPUTED_SCTR_LUT_DIM.w +
        float2(0.5,-0.5) / (PRECOMPUTED_SCTR_LUT_DIM.z*PRECOMPUTED_SCTR_LUT_DIM.w);
    f3UVW0.z =  (fQ0Slice + f4UVWQ.z) / PRECOMPUTED_SCTR_LUT_DIM.w;
    f3UVW0.z = clamp(f3UVW0.z, f2SliceMinMaxZ.x, f2SliceMinMaxZ.y);

    float fQ1Slice = min(fQ0Slice+1, PRECOMPUTED_SCTR_LUT_DIM.w-1);
    float fNextSliceOffset = (fQ1Slice - fQ0Slice) / PRECOMPUTED_SCTR_LUT_DIM.w;
    float3 f3UVW1 = f3UVW0 + float3(0,0,fNextSliceOffset);
    float3 f3Insctr0 = tex3DScatteringLUT.SampleLevel(samLinearClamp, f3UVW0, 0);
    float3 f3Insctr1 = tex3DScatteringLUT.SampleLevel(samLinearClamp, f3UVW1, 0);
    float3 f3Inscattering = lerp(f3Insctr0, f3Insctr1, fQWeight);

    return f3Inscattering;
}
```

Listing 1. Source code of the function performing one look-up into the pre-computed scattering texture.

The function first computes world space attributes of the current point, namely `fHeightAboveSurface` (h), `fCosViewZenithAngle` (c_α), `fCosSunZenithAngle` (c_φ), and `fCosSunViewAngle` (c_θ). It then calls `WorldParams2InsctrLUTCoords()` function (which implements equations (4)-(7)) to transform these attributes into the look-up table coordinates `f4UVWQ` ($u_h, u_\alpha, u_\varphi, u_\theta$). After that the function computes

interpolation weight for the fourth coordinate, performs two linear fetches from the 3D texture and interpolates the resulting values accordingly.

Handling of the u_α coordinate requires special care, because it has discontinuity at the horizon. Thus when performing look-ups into the scattering texture, it is very important that all the look-ups are consistent with respect to the horizon, otherwise artifacts will appear (Figure 5).

Figure 5. Artifacts at the horizon due to inconsistent look-ups.

To avoid the above problem, we must guarantee that that if the first look-up for the view ray is above (below) horizon, then all subsequent look-ups are also above (below) horizon. We use the previous texture coordinate to find out if the previous look-up was above or below horizon. For the first look-up, the texture coordinate is negative. The function given in Listing 2 computes u_α accounting for the previous look-up.

```
float ZenithAngle2TexCoord(float fCosZenithAngle, float fHeight,
                           float fTexDim, float power, float fPrevTexCoord)
{
    fCosZenithAngle = fCosZenithAngle;
    float fTexCoord;
    float fCosHorzAngle = GetCosHorizonAnlge(fHeight);
    // We use previous texture coordinate, if it is provided, to find out if previous look-up
    // was above or below horizon. If texture coordinate is negative, then this is the first
    // look-up
    bool bIsAboveHorizon = fPrevTexCoord >= 0.5;
    bool bIsBelowHorizon = 0 <= fPrevTexCoord && fPrevTexCoord < 0.5;
    if(  bIsAboveHorizon ||
        !bIsBelowHorizon && (fCosZenithAngle > fCosHorzAngle) )
    {
        // Scale to [0,1]
        fTexCoord = saturate( (fCosZenithAngle - fCosHorzAngle) / (1 - fCosHorzAngle) );
        fTexCoord = pow(fTexCoord, power);
        // Now remap texture coordinate to the upper half of the texture.
        // To avoid filtering across discontinuity at 0.5, we must map
        // the texture coordinate to [0.5 + 0.5/fTexDim, 1 - 0.5/fTexDim]
        fTexCoord = 0.5f + 0.5f / fTexDim + fTexCoord * (fTexDim/2 - 1) / fTexDim;
    }
    else
```

```
    {
        fTexCoord = saturate( (fCosHorzAngle - fCosZenithAngle) / (fCosHorzAngle - (-1)) );
        fTexCoord = pow(fTexCoord, power);
        // Now remap texture coordinate to the lower half of the texture.
        // To avoid filtering across discontinuity at 0.5, we must map
        // the texture coordinate to [0.5, 0.5 - 0.5/fTexDim]
        fTexCoord = 0.5f / fTexDim + fTexCoord * (fTexDim/2 - 1) / fTexDim;
    }

    return fTexCoord;
}
```

Listing 2. Computing texture coordinate u_α.

Computing Single Scattering

Pre-computation of single scattering radiance $L_{In}^{(1)}(C,\vec{v},\vec{l})$ for all possible positions C, view directions $\vec{v}$ and directions on sun $\vec{l}$ is implemented by a pixel shader given in Listing 3.

```
float3 PrecomputeSingleScatteringPS(SScreenSizeQuadVSOutput In) : SV_Target
{
    // Get attributes for the current point
    float2 f2UV = ProjToUV(In.m_f2PosPS);
    float fHeight, fCosViewZenithAngle, fCosSunZenithAngle, fCosSunViewAngle;
    InsctrLUTCoords2WorldParams(float4(f2UV, g_MiscParams.f2WQ),
                                fHeight, fCosViewZenithAngle,
                                fCosSunZenithAngle, fCosSunViewAngle );
    float3 f3EarthCentre = - float3(0,1,0) * EARTH_RADIUS;
    float3 f3RayStart = float3(0, fHeight, 0);
    float3 f3ViewDir = ComputeViewDir(fCosViewZenithAngle);
    float3 f3DirOnLight = ComputeLightDir(f3ViewDir, fCosSunZenithAngle, fCosSunViewAngle);

    // Intersect view ray with the top of the atmosphere and the Earth
    float4 f4Isecs;
    GetRaySphereIntersection2( f3RayStart, f3ViewDir, f3EarthCentre,
                               float2(EARTH_RADIUS, ATM_TOP_RADIUS),
                               f4Isecs);
    float2 f2RayEarthIsecs  = f4Isecs.xy;
    float2 f2RayAtmTopIsecs = f4Isecs.zw;

    if(f2RayAtmTopIsecs.y <= 0)
        return 0; // This is just a sanity check and should never happen
                  // as the start point is always under the top of the
                  // atmosphere (look at InsctrLUTCoords2WorldParams())

    // Set the ray length to the distance to the top of the atmosphere
    float fRayLength = f2RayAtmTopIsecs.y;
    // If ray hits Earth, limit the length by the distance to the surface
    if(f2RayEarthIsecs.x > 0)
        fRayLength = min(fRayLength, f2RayEarthIsecs.x);

    float3 f3RayEnd = f3RayStart + f3ViewDir * fRayLength;

    // Integrate single-scattering
    float3 f3Inscattering, f3Extinction;
    float fNumSteps = 100;
    IntegrateUnshadowedInscattering(f3RayStart,
                                    f3RayEnd,
                                    f3ViewDir,
```

```
                                    f3EarthCentre,
                                    f3DirOnLight,
                                    fNumSteps,
                                    f3Inscattering,
                                    f3Extinction);

    return f3Inscattering;
}
```

Listing 3. Shader implementation of single-scattering radiance pre-computation.

The shader first transforms input coordinates uvwq ($u_h, u_\alpha, u_\varphi, u_\theta$) into the world space attributes `fHeight` (h), `fCosViewZenithAngle` (c_α), `fCosSunZenithAngle` (c_φ), and `fCosSunViewAngle` (c_θ) by inverting equations (4)-(7). It then intersects the view ray with the top of the atmosphere or the Earth surface and computes the closest intersection point. Finally, it performs numerical integration of the single scattering along the ray.

Computing Scattered Light Radiance

Evaluation of equation (1) is performed by a pixel shader shown in Listing 4.

```
float3 ComputeSctrRadiancePS(SScreenSizeQuadVSOutput In) : SV_Target
{
    // Get attributes for the current point
    float2 f2UV = ProjToUV(In.m_f2PosPS);
    float fHeight, fCosViewZenithAngle, fCosSunZenithAngle, fCosSunViewAngle;
    InsctrLUTCoords2WorldParams( float4(f2UV, g_MiscParams.f2WQ), fHeight,
                                 fCosViewZenithAngle, fCosSunZenithAngle, fCosSunViewAngle );
    float3 f3EarthCentre = - float3(0,1,0) * EARTH_RADIUS;
    float3 f3RayStart = float3(0, fHeight, 0);
    float3 f3ViewDir = ComputeViewDir(fCosViewZenithAngle);
    float3 f3DirOnLight = ComputeLightDir(f3ViewDir, fCosSunZenithAngle, fCosSunViewAngle);

    // Compute particle density scale factor
    float2 f2ParticleDensity = exp( -fHeight / PARTICLE_SCALE_HEIGHT );

    float3 f3SctrRadiance = 0;
    // Go through a number of samples randomly distributed over the sphere
    for(int iSample = 0; iSample < NUM_RANDOM_SPHERE_SAMPLES; ++iSample)
    {
        // Get random direction
        float3 f3RandomDir = normalize( g_tex2DSphereRandomSampling.Load(int3(iSample,0,0)) );
        // Get the previous order in-scattered light when looking in direction f3RandomDir
        // (the light thus goes in direction -f3RandomDir)
        float4 f4UVWQ = -1;
        float3 f3PrevOrderSctr =
             LookUpPrecomputedScattering(f3RayStart, f3RandomDir, f3EarthCentre,
                                         f3DirOnLight.xyz, g_tex3DPreviousSctrOrder, f4UVWQ);

        // Apply phase functions for each type of particles
        // Note that total scattering coefficients are baked into the angular
        // scattering coeffs
        float3 f3DRlghInsctr = f2ParticleDensity.x * f3PrevOrderSctr;
        float3 f3DMieInsctr  = f2ParticleDensity.y * f3PrevOrderSctr;
        float fCosTheta = dot(f3ViewDir, f3RandomDir);
        ApplyPhaseFunctions(f3DRlghInsctr, f3DMieInsctr, fCosTheta);

        f3SctrRadiance += f3DRlghInsctr + f3DMieInsctr;
```

```
    }
    // Since we tested N random samples, each sample covered 4*Pi / N solid angle
    // Note that our phase function is normalized to 1 over the sphere. For instance,
    // uniform phase function would be p(theta) = 1 / (4*Pi).
    // Notice that for uniform intensity I if we get N samples, we must obtain exactly I after
    // numeric integration
    return f3SctrRadiance * 4*PI / NUM_RANDOM_SPHERE_SAMPLES;
}
```

Listing 4. Pixel shader for computing order *n* scattered light radiance.

Like the previous one, the shader first transforms input coordinates uvwq into the world space attributes by inverting equations (4)-(7). It then computes Rayleigh and Mie particle density scale factors `f2ParticleDensity` for the current point. After that it samples `NUM_RANDOM_SPHERE_SAMPLES` random directions, loads previous order scattering radiance `f3PrevOrderSctr` from each direction, computes integrand of equation (1) and accumulates it in `f3SctrRadiance` variable. The resulting net radiance is then multiplied by $4\pi / N$ where N is the number of samples, as each sample covers 1/N fraction of the entire sphere, which is 4π.

Computing Multiple Scattering Order

The shader given in Listing 5 implements numerical integration of equation (2).

```
float3 ComputeScatteringOrderPS(SScreenSizeQuadVSOutput In) : SV_Target
{
    // Get attributes for the current point
    float2 f2UV = ProjToUV(In.m_f2PosPS);
    float fHeight, fCosViewZenithAngle, fCosSunZenithAngle, fCosSunViewAngle;
    InsctrLUTCoords2WorldParams(float4(f2UV, g_MiscParams.f2WQ), fHeight, fCosViewZenithAngle,
                                fCosSunZenithAngle, fCosSunViewAngle );
    float3 f3EarthCentre = - float3(0,1,0) * EARTH_RADIUS;
    float3 f3RayStart = float3(0, fHeight, 0);
    float3 f3ViewDir = ComputeViewDir(fCosViewZenithAngle);
    float3 f3DirOnLight = ComputeLightDir(f3ViewDir, fCosSunZenithAngle, fCosSunViewAngle);

    // Intersect the ray with the atmosphere and Earth
    float4 f4Isecs;
    GetRaySphereIntersection2( f3RayStart, f3ViewDir, f3EarthCentre,
                               float2(EARTH_RADIUS, ATM_TOP_RADIUS),
                               f4Isecs);
    float2 f2RayEarthIsecs  = f4Isecs.xy;
    float2 f2RayAtmTopIsecs = f4Isecs.zw;

    if(f2RayAtmTopIsecs.y <= 0)
        return 0; // This is just a sanity check and should never happen
                  // as the start point is always under the top of the
                  // atmosphere (look at InsctrLUTCoords2WorldParams())

    float fRayLength = f2RayAtmTopIsecs.y;
    if(f2RayEarthIsecs.x > 0)
        fRayLength = min(fRayLength, f2RayEarthIsecs.x);

    float3 f3RayEnd = f3RayStart + f3ViewDir * fRayLength;

    const float fNumSamples = 64;
    float fStepLen = fRayLength / fNumSamples;

    float4 f4UVWQ = -1;
```

```
    float3 f3PrevSctrRadiance =
         LookUpPrecomputedScattering(f3RayStart, f3ViewDir, f3EarthCentre, f3DirOnLight.xyz,
                                     g_tex3DPointwiseSctrRadiance, f4UVWQ);
    float2 f2PrevParticleDensity = exp( -fHeight / PARTICLE_SCALE_HEIGHT );

    float2 f2NetParticleDensityFromCam = 0;
    float3 f3Inscattering = 0;

    for(float fSample=1; fSample <= fNumSamples; ++fSample)
    {
        float3 f3Pos = lerp(f3RayStart, f3RayEnd, fSample/fNumSamples);

        float fCurrHeight = length(f3Pos - f3EarthCentre) - EARTH_RADIUS;
        float2 f2ParticleDensity = exp( -fCurrHeight / PARTICLE_SCALE_HEIGHT );

        f2NetParticleDensityFromCam +=
                   (f2PrevParticleDensity + f2ParticleDensity) * (fStepLen / 2.f);
        f2PrevParticleDensity = f2ParticleDensity;

        // Get optical depth
        float3 f3RlghOpticalDepth =
                g_MediaParams.f4RayleighExtinctionCoeff.rgb * f2NetParticleDensityFromCam.x;
        float3 f3MieOpticalDepth  =
                g_MediaParams.f4MieExtinctionCoeff.rgb      * f2NetParticleDensityFromCam.y;

        // Compute extinction from the camera for the current integration point:
        float3 f3ExtinctionFromCam = exp( -(f3RlghOpticalDepth + f3MieOpticalDepth) );

        // Get attenuated scattered light radiance in the current point
        float4 f4UVWQ = -1;
        float3 f3SctrRadiance = f3ExtinctionFromCam *
                LookUpPrecomputedScattering(f3Pos, f3ViewDir, f3EarthCentre, f3DirOnLight.xyz,
                                            g_tex3DPointwiseSctrRadiance, f4UVWQ);
        // Update in-scattering integral
        f3Inscattering += (f3SctrRadiance +  f3PrevSctrRadiance) * (fStepLen/2.f);
        f3PrevSctrRadiance = f3SctrRadiance;
    }

    return f3Inscattering;
}
```

Listing 5. Pixel shader for computing order *n* in-scattered light radiance.

Like the shader given in Listing 3, this shader first transforms input coordinates uvwq into the world space attributes and reconstructs the view ray from camera to the top of the atmosphere or intersection with the Earth, whichever is closer. It then performs trapezoidal integration of equation (2). At each step the shader updates extinction from the camera `f3ExtinctionFromCam` and computes differential amount of light `f3SctrRadiance` in-scattered from the unit length ray section at current point. The total in-scattered light is accumulated in `f3Inscattering`.

Computing Ambient Sky Light

Pre-computation of the ambient sky light according to equation (3) is performed by the shader shown in Listing 6.

```
float3 PrecomputeAmbientSkyLightPS(SScreenSizeQuadVSOutput In) : SV_Target
{
    float fU = ProjToUV(In.m_f2PosPS).x;
```

```
    float3 f3RayStart = float3(0,20,0);
    float3 f3EarthCentre =  -float3(0,1,0) * EARTH_RADIUS;
    float fCosZenithAngle = clamp(fU * 2 - 1, -1, +1);
    float3 f3DirOnLight =
            float3(sqrt(saturate(1 - fCosZenithAngle*fCosZenithAngle)), fCosZenithAngle, 0);
    float3 f3SkyLight = 0;
    // Go through a number of random directions on the sphere
    for(int iSample = 0; iSample < NUM_RANDOM_SPHERE_SAMPLES; ++iSample)
    {
        // Get random direction
        float3 f3RandomDir = normalize( g_tex2DSphereRandomSampling.Load(int3(iSample,0,0)) );
        // Reflect directions from the lower hemisphere
        f3RandomDir.y = abs(f3RandomDir.y);
        // Get multiple scattered light radiance when looking in direction f3RandomDir
        // (the light thus goes in direction -f3RandomDir)
        float4 f4UVWQ = -1;
        float3 f3Sctr =
                LookUpPrecomputedScattering(f3RayStart, f3RandomDir, f3EarthCentre,
                                            f3DirOnLight.xyz, g_tex3DPreviousSctrOrder,
                                            f4UVWQ);
        // Accumulate ambient irradiance through the horizontal plane
        f3SkyLight += f3Sctr * dot(f3RandomDir, float3(0,1,0));
    }
    // Each sample covers 2 * PI / NUM_RANDOM_SPHERE_SAMPLES solid angle
    // (integration is performed over upper hemisphere)
    return f3SkyLight * 2 * PI / NUM_RANDOM_SPHERE_SAMPLES;
}
```

Listing 6. Pre-computing ambient sky light.

The shader first computes cosine of the sun zenith angle `fCosZenithAngle` and reconstructs sun light direction. It then samples `NUM_RANDOM_SPHERE_SAMPLES` random directions on the upper hemisphere, loads multiple scattering radiance intensity from each direction, computes integrand of equation (3) and accumulates it in `f3SkyLight` variable. The resulting irradiance is then multiplied by $2\pi / N$ where N is the number of samples, as each sample covers $1/N$ fraction of the upper hemisphere, which is 2π.

Light Shafts Calculation Using Scattering Look-up Table

Since multiple scattering radiance $L_{In}^{M}(C,\vec{v},\vec{l})$ is pre-computed, we can evaluate in-scattering contribution between any two points P_1 and P_2 on the view ray using the following equation:

$$L_{In}(P_1,P_2,\vec{l}) = e^{-T(C\to P_1)}L_{In}^{M}(P_1,\vec{v},\vec{l}) - e^{-T(C\to P_2)}L_{In}^{M}(P_2,\vec{v},\vec{l}). \quad (8)$$

Evaluating the above equation at each step of the ray marching loop would be prohibitively expensive. In original method Bruneton and Neyret compute total length of the ray fraction which is lit by the sun and then apply equation (8) assuming that the fraction is continuous and starts directly at the camera [Bruneton and Neyret 08]. When camera is located in shadow (Figure 6, left) this method gives too bright scattering.

In our method we take advantage of the ray marching process not only to evaluate total lit length D_{Lit}, but also to compute distance D_{Start} to the first lit point on the view ray (Figure 6).

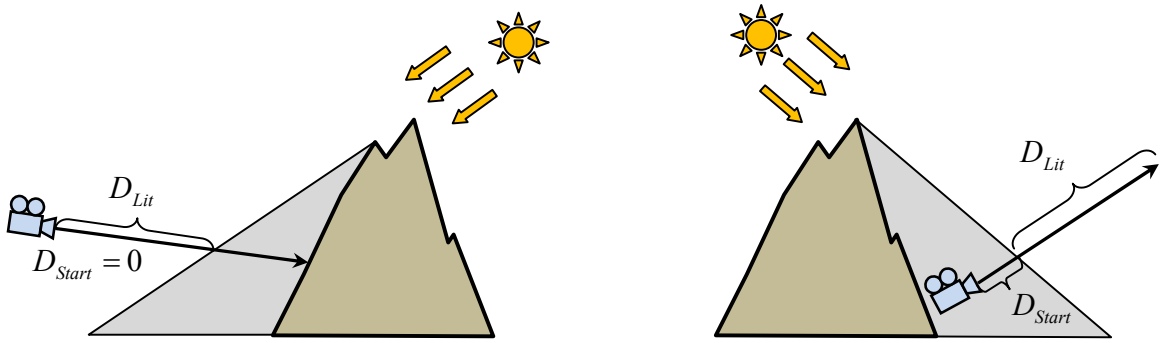


Figure 6. Total length D_{Lit} of the illuminated ray fraction and distance D_{Start} to the first lit point.

After the ray marching loop is completed, we use D_{Lit} and D_{Start} to compute start and end points of the lit ray fraction and apply equation (8) as show in the code snippet below:

```
float3 f3LitSectionEnd = f3LitSectionStart + fTotalLitLength * f3ViewDir;

float3 f3ExtinctionToStart =
  GetExtinctionUnverified(f3RestrainedCameraPos, f3LitSectionStart, f3ViewDir, f3EarthCentre);
float4 f4UVWQ = -1;
f3MultipleScattering = f3ExtinctionToStart *
    LookUpPrecomputedScattering(f3LitSectionStart, f3ViewDir, f3EarthCentre,
                                g_LightAttribs.f4DirOnLight.xyz, tex3DSctrLUT, f4UVWQ);

float3 f3ExtinctionToEnd =
  GetExtinctionUnverified(f3RestrainedCameraPos, f3LitSectionEnd, f3ViewDir, f3EarthCentre);
f3MultipleScattering -= f3ExtinctionToEnd *
    LookUpPrecomputedScattering(f3LitSectionEnd, f3ViewDir, f3EarthCentre,
                                g_LightAttribs.f4DirOnLight.xyz, tex3DSctrLUT, f4UVWQ);

f3InsctrIntegral += max(f3MultipleScattering, 0);
```

In the code fragment above, `GetExtinctionUnverified()` function evaluates extinction between two points without checking if the points are inside the atmosphere. `f3RestrainedCameraPos` variable contains camera position restricted by the top of the atmosphere. Important aspect to notice is that texture coordinates `f4UVWQ` returned from the first call to `LookUpPrecomputedScattering()` function are passed to the second call to assure consistent sampling with respect to horizon.

Occluded vs unoccluded higher order scattering

Higher order scattering is very difficult to model precisely because the path a photon can reach the camera can be very complicated. Some simplifications are necessary to make multiple scattering feasible for real-time rendering. Bruneton and Neyret in their work account for multiple scattering contribution from the same illuminated ray fraction only [Bruneton and Neyret 08]. The resulting sky could look too dark when camera is located in Earth's shadow, which is especially noticeable during the twilight (Figure 7, left top). We believe that more plausible results could be obtained if higher order scattering is considered unoccluded and is thus always added to the resulting scattering radiance. The reasoning behind this approach is that higher order scattering is caused by the radiance coming from the whole

hemisphere. With the exception of some cases, usually most of the points on the ray are not occluded from the sky dome radiance, making significant contribution to higher order scattering. The image rendered with this method looks significantly more realistic (Figure 7, right top). To implement this approach, we use two look-up tables. First, after completing the ray marching, we perform two look-ups into the single scattering look-up table (`tex3DSingleScattering`) as described above. After that we do another two look-ups into the higher order look-up table (`tex3DHighOrderScattering`) to get unoccluded higher order scattering contribution from the whole ray.

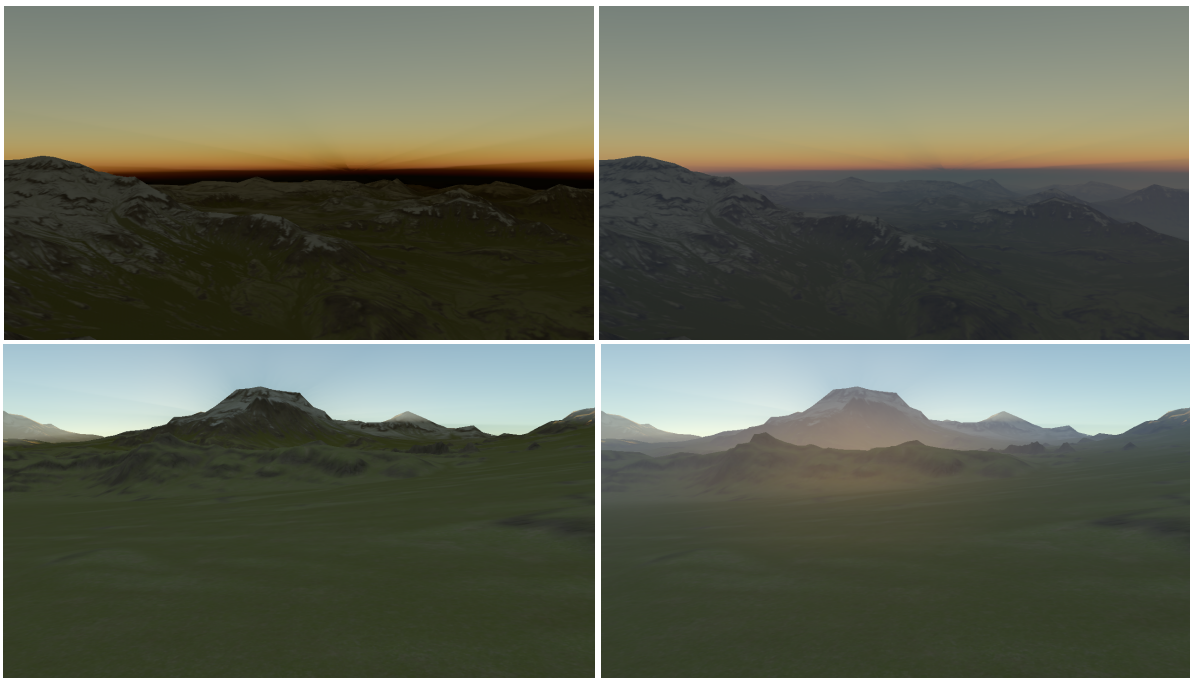


Figure 7. Occluded higher order scattering (left column) vs unoccluded higher order scattering (right column).

Though the method described above improves twilight sky dramatically, it results in some false glow in the sun direction when sun is actually occluded (Figure 7, bottom row). We however believe that this minor drawback can be treaded for the improved twilight sky.

Terrain shaders

Terrain shaders compute sun light extinction due to the atmosphere and account for the ambient sky light. Sun light extinction and ambient sky light are evaluated in the vertex shader as shown in the following code snippet:

```
void GetSunLightExtinctionAndSkyLight(in float3 f3PosWS,
                                      out float3 f3Extinction,
                                      out float3 f3AmbientSkyLight)
{
    float3 f3EarthCentre = float3(0, -g_MediaParams.fEarthRadius, 0);
    float3 f3DirFromEarthCentre = f3PosWS - f3EarthCentre;
    float fDistToCentre = length(f3DirFromEarthCentre);
```

```
    f3DirFromEarthCentre /= fDistToCentre;
    float fHeightAboveSurface = fDistToCentre - g_MediaParams.fEarthRadius;
    float fCosZenithAngle = dot(f3DirFromEarthCentre, g_LightAttribs.f4DirOnLight.xyz);

    float fRelativeHeightAboveSurface = fHeightAboveSurface / g_MediaParams.fAtmTopHeight;
    float2 f2ParticleDensityToAtmTop =
        g_tex2DOccludedNetDensityToAtmTop.SampleLevel(samLinearClamp,
            float2(fRelativeHeightAboveSurface, fCosZenithAngle*0.5+0.5), 0).xy;

    float3 f3RlghOpticalDepth =
            g_MediaParams.f4RayleighExtinctionCoeff.rgb * f2ParticleDensityToAtmTop.x;
    float3 f3MieOpticalDepth  =
            g_MediaParams.f4MieExtinctionCoeff.rgb      * f2ParticleDensityToAtmTop.y;

    // And total extinction for the current integration point:
    f3Extinction = exp( -(f3RlghOpticalDepth + f3MieOpticalDepth) );

    f3AmbientSkyLight =
             g_tex2DAmbientSkylight.SampleLevel( samLinearClamp,
                                                 float2(fCosZenithAngle*0.5+0.5, 0.5), 0);
}
```

Interpolated extinction and ambient sky light are then passed to the pixel shader, which uses them to attenuate sun light intensity and approximate ambient light as shown in the following code snippet:

```
// Attenuate extraterrestrial sun color with the extinction factor
float3 f3SunLight = g_LightAttribs.f4ExtraterrestrialSunColor.rgb * In.f3SunLightExtinction;
// Ambient sky light is not pre-multiplied with the sun intensity
float3 f3AmbientSkyLight =
        g_LightAttribs.f4ExtraterrestrialSunColor.rgb * In.f3AmbientSkyLight;
// Account for occlusion by the ground plane
f3AmbientSkyLight *= saturate((1 + dot(EarthNormal, f3Normal))/2.f);
```

Tone mapping

The sample supports a number of different tone mapping operators, namely:

- **Exp** – simple exponential tone mapping operator;
- **Reinhard** – basic Reinhard tone mapping operator [Reinhard et al 02];
- **Modified Reinhard** – modified Reinhard tone mapping operator [Reinhard et al 02];
- **Uncharted 2** – tone mapping operator from Uncharted 2 [Hable 10];
- **Filmic ALU** – filmic tone mapping operator [Hable 10];
- **Logarithmic** – logarithmic tone mapping operator [Drago et al 03];
- **Adaptive log** – adaptive logarithmic tone mapping operator [Drago et al 03].

To estimate average scene luminance, we unwarp the scene into the low-resolution (64x64) buffer, outputting luminance only, and generate the whole mipmap chain. The coarsest level then contains average value.

Performance Results and Discussion

Performance of different stages of the algorithm on Intel HD graphics 5000 is given in the Figure 8. Four different modes were evaluated:

- **SS-Int** – single scattering computed with the numerical integration, no higher order scattering;
- **SS-Int+MS-UnO** – single scattering computed with the numerical integration plus unoccluded higher order scattering;
- **SS-LUT+MS-UnO** – single scattering plus unoccluded higher order scattering both computed using two look-up tables;
- **SS-LUT+MS-Occ** – single and occluded higher order scattering computed with the two look-ups into single table.

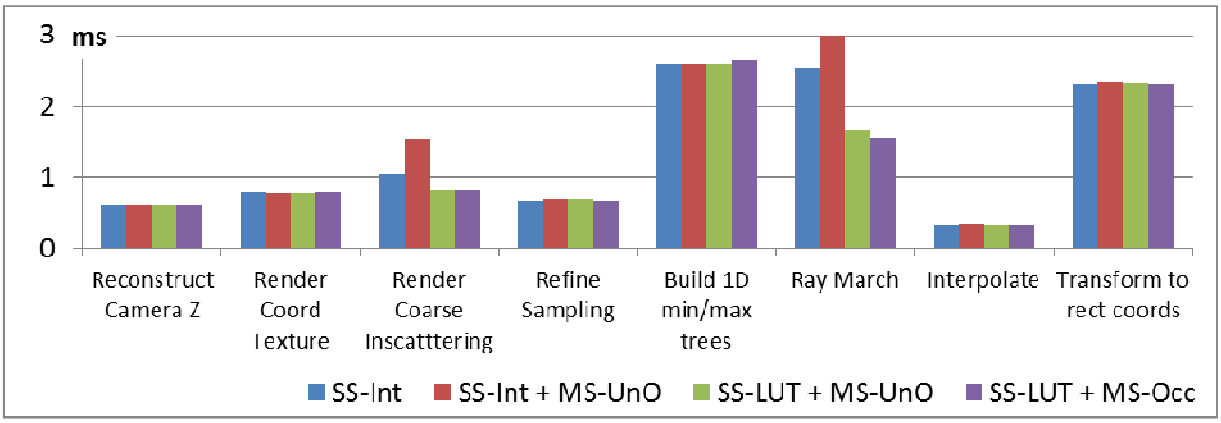


Figure 8. Performance of different stages of the algorithm on Intel HD graphics 5000 (1280x720 resolution).

It can be seen that most of the stages take the same time in all modes except for the rendering coarse inscattering and the ray marching. Both stages take the most time in **SS-Int+MS-UnO** mode because in this case besides evaluating single scattering integral with the numerical integration, additional look-up into the scattering look-up table is performed. With this additional look-up, ray marching becomes 24% slower (3.17 ms vs 2.55 ms). Meanwhile, overall slowdown due adding higher-order scattering is modest 10.1%, rising the total time from 11.3 to 12.4 ms. It must be noted that this mode generates the most convincing visual results, so additional performance cost seems to be justified.

In the two remaining modes, **SS-LUT+MS-UnO** and **SS-LUT+MS-Occ**, numerical integration of single scattering is not performed and only total lit length and distance to the first point in light is computed. This makes ray marching loop significantly simpler and faster, as Figure 8 reveals. It can be seen that the difference between two modes is in fact insignificant despite the fact that in SS-LUT+MS-UnO mode, four look-ups into two tables are performed, while two look-ups into one table are performed in SS-LUT+MS-Occ mode. The ray marching time goes down by 36% from 2.5 ms to 1.6 ms. Computing coarse inscattering is also much faster in this case, because instead of performing trapezoidal integration, only two look-ups into the table are performed. The time goes down from 1.05 ms to 0.81 ms. Overall speed-up compared to numerical integration only (SS-Int mode) is 10% (11.3 ms vs 10.0 ms).

Controls

- **w**,**s**,**a**,**d**,**q**,**e** - move camera

- **shift** – accelerate
- **left mouse button** – rotate camera
- **right mouse button** – rotate light
- **F1** – show/hide help text
- **F8** – rebuild shaders (could be used to modify shaders and see the effect at run time)

GUI

The sample GUI has three panels, which can be selected using the top most drop-down list: **basic**, **additional** and **tone mapping**.

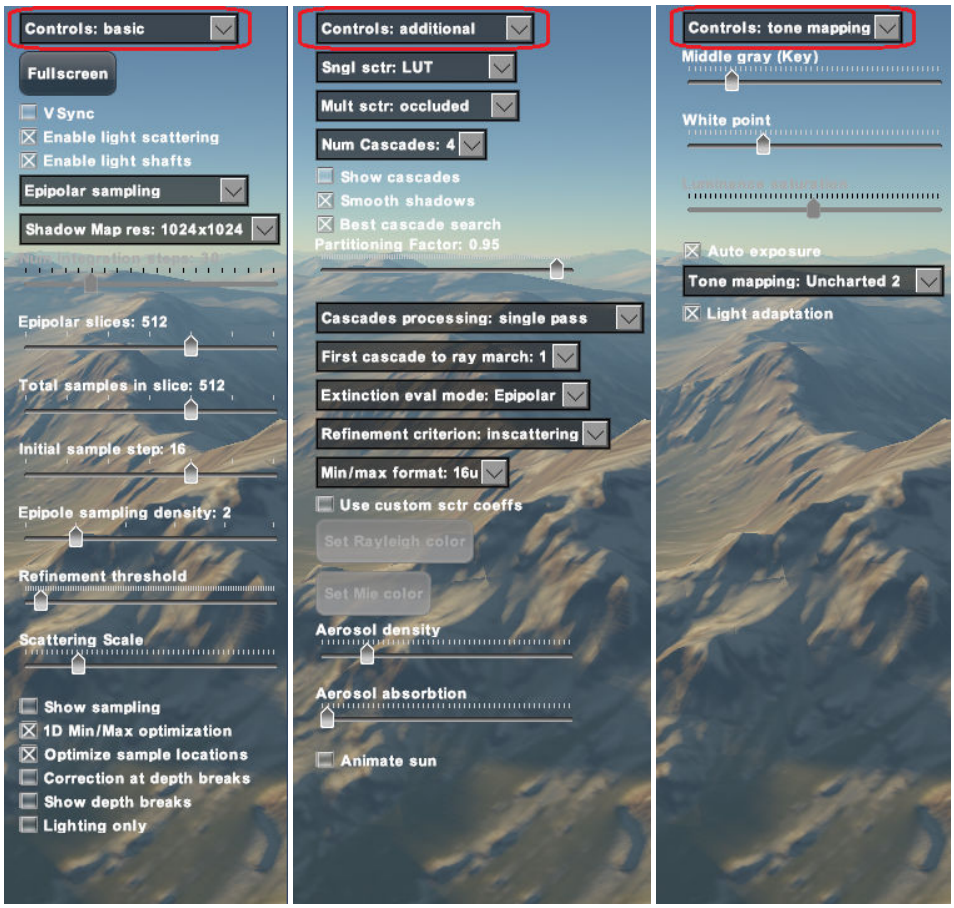
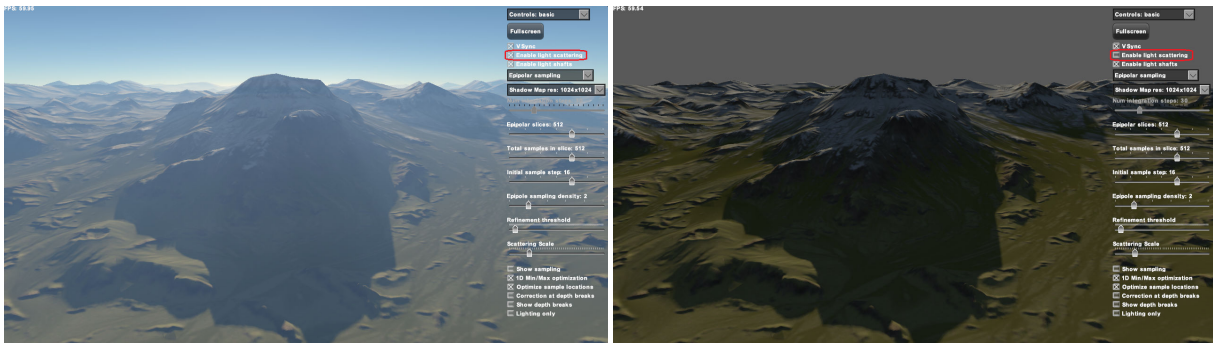


Figure 4. Left to right: basic, additional and tone mapping control sets.

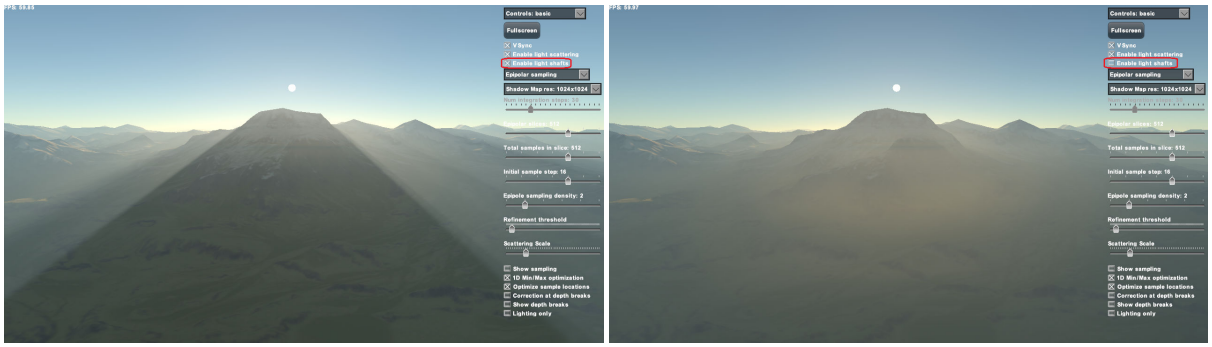
Basic controls

- **Full screen** button – toggles the full screen mode.
- **VSync** checkbox – toggles vertical synchronization at 60 fps.

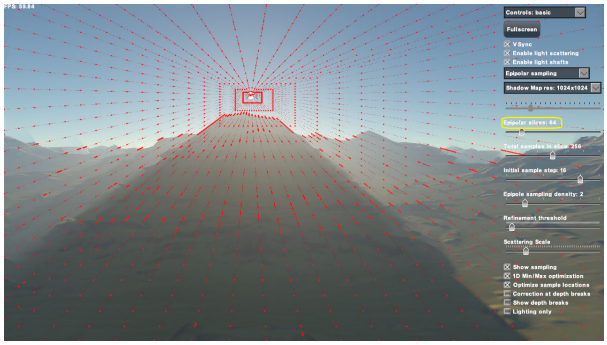
- **Enable light scattering** checkbox – enables or disables the light scattering effect.



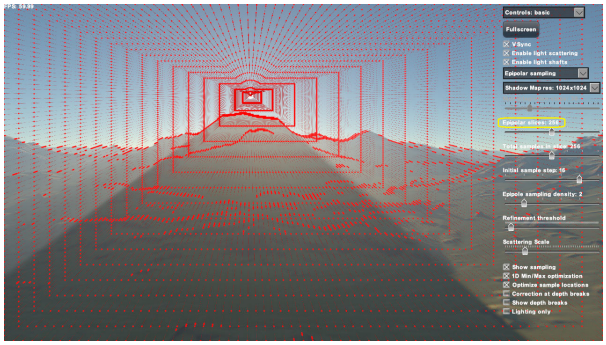
- **Enable light shafts** checkbox – enables or disables rendering shafts of light. If this option is disabled, simpler algorithm is used to compute scattering integral which ignores shadows.



- **Scattering technique** selection drop-down list has two options:
    - **Epipolar sampling** – calculate scattering effect using epipolar sampling;
    - **Brute force ray marching** – calculate scattering effect for each screen pixel without 1D min/max binary tree optimization. This mode can be used to generate the reference "ground truth" image and compare the quality with the epipolar sampling.
- **Shadow map resolution** drop down list enables selecting the resolution from 512x512 up to 4096x4096. Note that the number of cascades can be selected on the **Additional** controls panel.
- **Num integration steps** slider sets the number of steps performed when computing single scattering integral. This control is only enabled when **Enable light shafts** checkbox is not marked and when **Single sctr: integration** is selected on the **Additional** controls panel.
- **Epipolar slices** slider selects the number of slices in the range from 32 to 2048. The more epipolar slice, the higher visual quality and the higher computational cost. Good visual results can be obtained when number of slices is at least half the maximum screen resolution (for 1280x720 resolution, good results are obtained for 512-1024 slices).



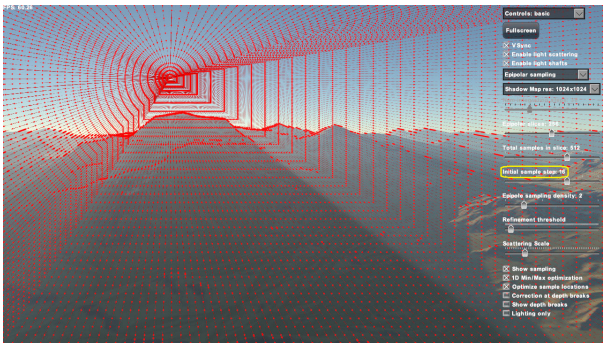
64 slices



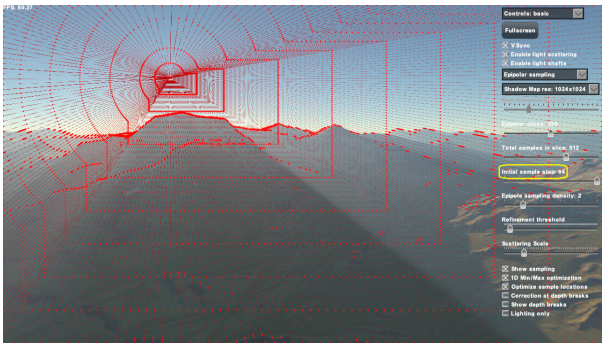
256 slices

- **Total samples in slice** slider enables selecting the number of samples in the range from 32 to 2048. Convincing visual results are generated when number of samples is at least half the maximum screen resolution (for 1280x720 resolution, good results are obtained for 512-1024 samples).

- **Initial sample step** slider sets the step for the initial ray marching samples. The lower the step, the more initial ray marching samples will be placed along each epipolar line and the higher image quality will be obtained. Initial ray marching samples are seen as rectangles/circles going away from the epipole at regular steps.



Initial step: 16



Initial step: 64

- **Epipole sampling density** slider controls additional increase of the initial ray marching sampling density near the epipole, which could be required to account for high frequency light variations due to strong forward Mie scattering.

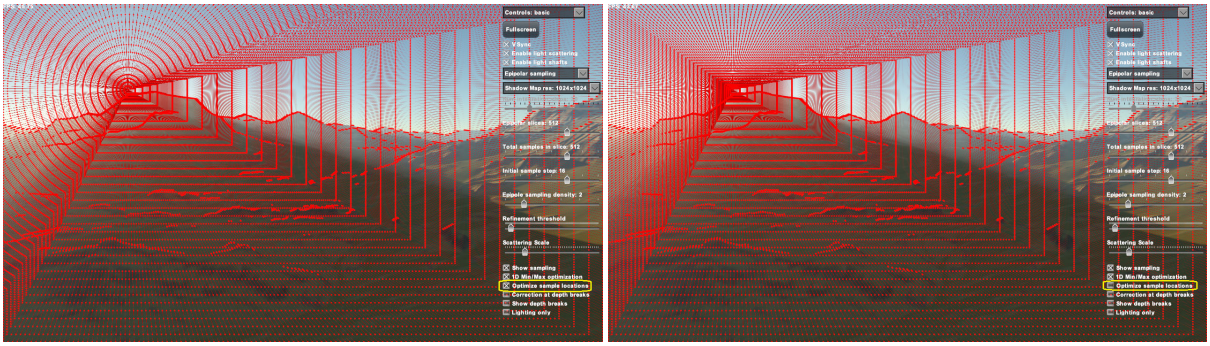
- **Refinement threshold** slider controls the refinement accuracy. Smaller threshold causes more ray marching samples to be placed at discontinuities of light intensity and produces higher quality rendering.

- **Scattering scale** slider controls the intensity of the scattering effect (note that this option has almost no effect when tone mapping with automatic exposure is enabled).

- **Show sampling** check box toggles visualization of the epipolar sampling structure. Thick red dots denote ray marching samples, thin red dots represent interpolation samples.

- **1D Min/Max optimization** check box enables/disables using 1-D min/max binary trees to accelerate ray marching.

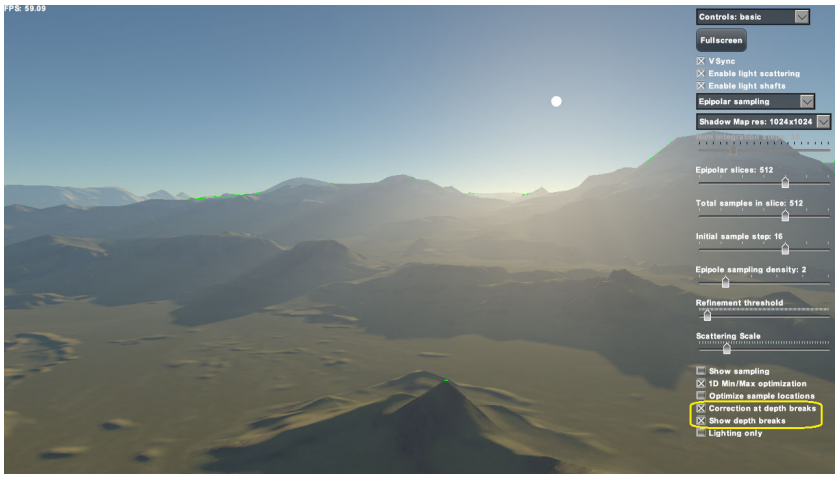
- **Optimize sample locations** checkbox enables an option to fix oversampling along short epipolar lines.



Optimized locations Non-optimized locations

- **Correction at depth breaks** checkbox toggles additional fix-up pass for correcting these samples, for which scattered light cannot be precisely computed by unwarping epipolar scattering image. During this pass, ray marching algorithm is executed for these pixels. 1D min/max binary tree acceleration cannot be used at this time, because the pixels do not belong to any epipolar line. Thus ray marching is much less efficient in this case. For typical terrain scene, usually only a very few pixels (if any) require correction (see picture below). So this option can be safely disabled without noticeable loss of visual quality.

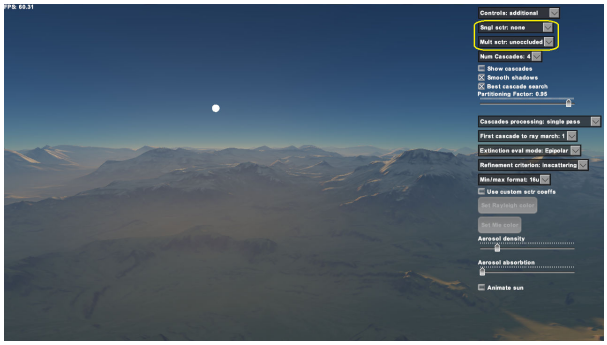
- **Show depth breaks** checkbox enables visualizing (in green) these pixels, for which correction pass is performed. This option has effect only when **Correction at depth breaks** checkbox is marked.



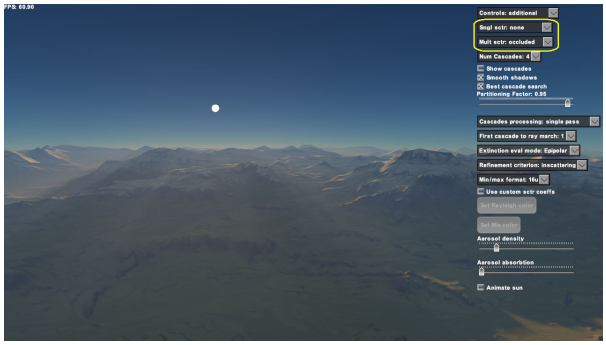
- **Lighting only** checkbox toggles showing the light scattering effects only.

Additional controls

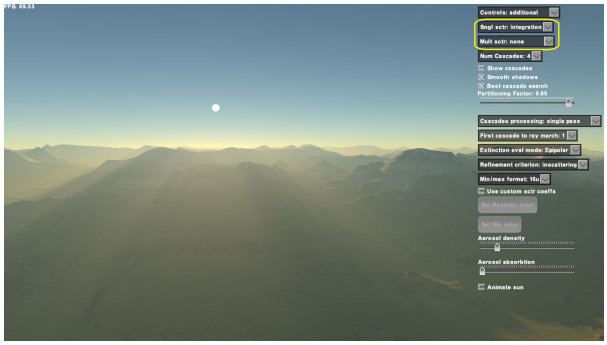
- **Single scattering mode** drop-down list has the following options:
    - **Singl sctr: none** – do not compute single scattering.
    - **Singl sctr: integration** – compute single scattering with the numerical integration during ray marching.
    - **Singl sctr: LUT** – compute single scattering using look-up table. If this option is enabled, only total length of the illuminated portion of the ray is evaluated during ray marched and distance to the first lit section is stored. After that two look-ups are performed to get single scattering contribution along the ray.
- **Multiple scattering mode** drop-down list has the following options:
    - **Mult sctr: none** – do not compute multiple scattering.
    - **Mult sctr: occluded** – account for multiple scattering contribution only from the illuminated part of the ray. This is implemented by performing look-ups into the higher order scattering texture, if single scattering mode is **none** or **integration**. If single scattering mode is LUT, then single and higher order scattering are computed by performing look-ups into the multiple scattering look-up texture.
    - **Mult sctr: unoccluded** – add higher order scattering contribution from the whole view ray. This is implemented by performing two look-ups into the higher order scattering look-up texture.



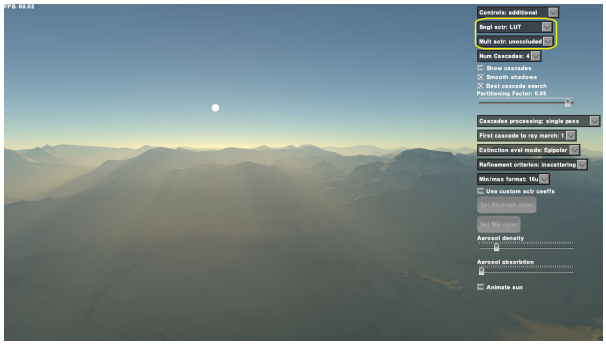
SS-none + MS-unoccluded



SS-none + MS-occluded

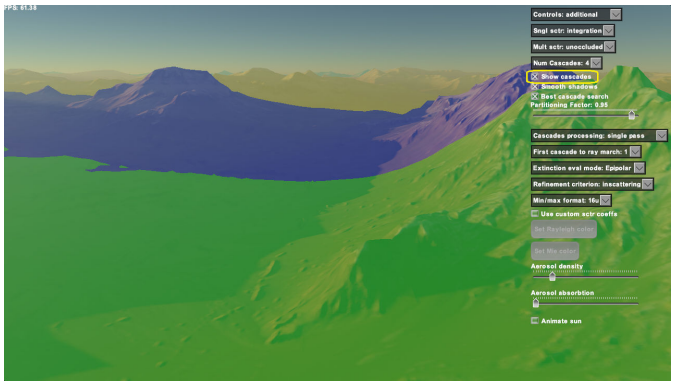


SS-integration + MS-none



SS-LUT + MS-unoccluded

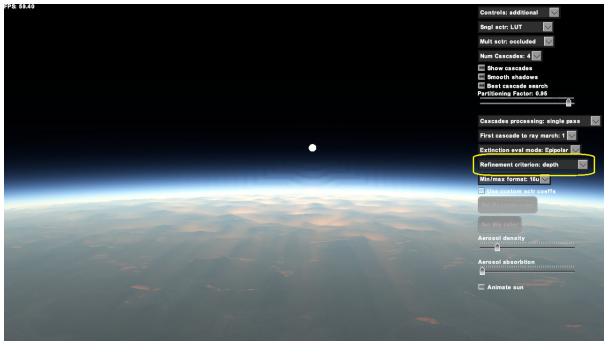
- **Num cascades** drop down enables selecting the number of shadow cascades in the range from 1 to 8.
- **Show cascades** checkbox toggles visualization of shadow map cascades.



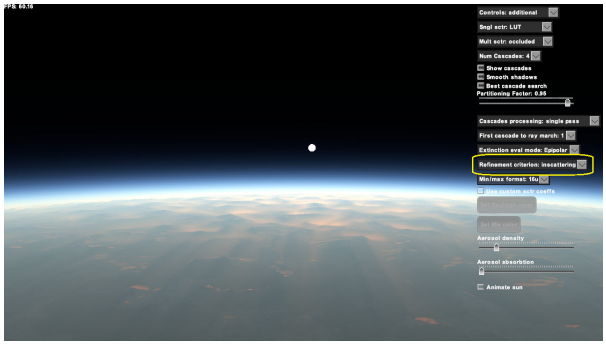
- **Smooth shadows** checkbox enables softening shadow edges by averaging 5 PCF samples.
- **Best cascade search** checkbox enables improving shadow quality by finding the smallest cascade the point projects into instead of relying on cascade z range.
- **Partitioning factor** slider controls the ratio between linear and logarithmic cascade distribution.
- **Cascade processing mode** drop down contains the following options:

    - **Single pass** – all cascades are processed by a single draw call. The shader loops through all the cascades;
    - **Multi pass** – each cascade is processed by a separate draw call. The scattering contribution from each cascade is accumulated in a scattering texture using alpha blending;
    - **Multi pass inst** – works in the same way as Multi pass, but single instanced draw call is issued.

- **First cascade to ray march** dropdown enables selecting the smallest cascade which will be considered during ray marching. Usually one or two smallest cascades can be safely skipped without affecting visual quality.
- **Extinction evaluation mode** dropdown enables selecting one of the following two modes:
    - **Epipolar** – render epipolar extinction texture during coarse scattering integral calculation pass and use it at the final unwarping stage to get extinction for each pixel.
    - **Per pixel** – use analytical expression to evaluate extinction individually for each pixel. Selecting one of these two options does not affect visual quality. Epipolar mode however is much faster.
- **Refinement criterion** dropdown enables selecting one of the following two inscattering refinement criterions:
    - **Depth** – use difference in depth as a refinement criterion.
    - **Inscattering** – use difference in coarse unshadowed inscattering integral as the refinement criterion. This criterion is much more appropriate than depth criterion as significant differences in scattered light is what we need to find to refine sampling. The advantages of this criterion become apparent for outer space views where depth does not exhibit any changes while scattered light varies significantly. As a result, depth criterion generates evident banding artifacts near epipole, while inscattering criterion does not suffer from this problem.



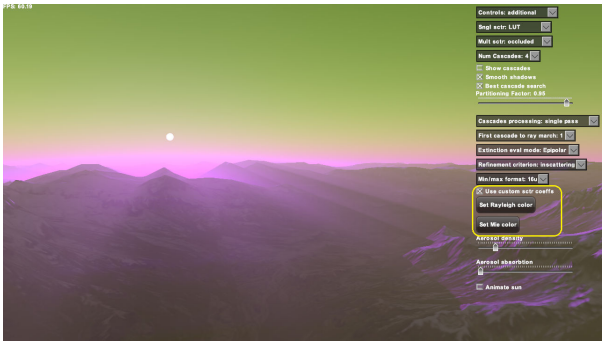
Refinement criterion - depth



Refinement criterion - inscattering

- **Min/max format** drop down enables selecting format of the texture storing 1D min/max binary trees. Two options are available: 16-bit unorm and 32-bit float. This option has no visual effect, but selecting 16-bit unorm format improves performance a bit.

- **Use custom sctr coeffs** check box enables setting custom Rayleigh and Mie scattering coefficients. This option enables **Set Rayleigh color** and **Set Mie Color** buttons which selects corresponding scattering coefficients. Note that changing one of these options takes some time because all the look-up tables need to be re-computed.



- **Aerosol density** and **Aerosol absorption** sliders controls the appropriate parameters. Note that changing either parameter takes some time because all the look-up tables need to be re-computed.

- **Animate sun** checkbox toggles sun animation.

Tone mapping controls

- **Middle gray (Key)** slider controls the tone of the scene. Default value is 0.18. Larger values make the scene look brighter, smaller values make it look dimmer.

- **White point** slider selects the white point value for some tone mapping operators (Reinhard Mod, Uncharted 2, Logarithmic and Adaptive log).

- **Luminance saturation** slider controls additional color saturation for Exp, Reinhard, Reinhard mod, Logarithmic and Adaptive log tone mapping operators.

- **Auto exposure** enables computing logarithmic average luminance of the scene to automatically adjust exposure.

- **Tone mapping operator** provides selection of one of the following modes:

    - **Exp** – exponential tone mapping operator;
    - **Reinhard** – basic Reinhard tone mapping operator;
    - **Reinhard mod** – modified Reinhard tone mapping operator;
    - **Uncharted 2** – tone mapping operator from Uncharted 2;
    - **Filmic ALU** – filmic tone mapping operator;
    - **Logarithmic** – logarithmic tone mapping operator;
    - **Adaptive log** – adaptive logarithmic tone mapping operator.

- **Light adaptation** checkbox enables simulation of temporal light adaptation.

References

[Bruneton and Neyret 08] Eric Bruneton and Fabrice Neyret. “Precomputed Atmospheric Scattering.” *Comput. Graph. Forum* 27:4 (2008), 1079-1086. Special Issue: Proceedings of the 19th Eurographics Symposium on Rendering 2008.

[Reinhard et al 02] E. Reinhard, M. Stark, P. Shirley, and J. Ferwerda. “Photographic Tone Reproduction for Digital Images”. *ACM Transactions on Graphics*, 21(3):267–276, 2002. Available online (http://www.cs.utah.edu/~reinhard/cdrom/tonemap.pdf).

[Drago et al 03] F. Drago, K. Myszkowski, T. Annen, and N. Chiba. “Adaptive logarithmic mapping for displaying high contrast scenes”. *Computer Graphics Forum*, vol. 22, no. 3, 2003. Available online (http://www.mpi-inf.mpg.de/resources/tmo/logmap/logmap.pdf).

[Hable 10] John Hable. “Uncharted 2: HDR Lighting" In Proceedings Game Developer Conference, 2010. Available online (http://www.gdcvault.com/play/1012459/Uncharted_2__HDR_Lighting).

Notices