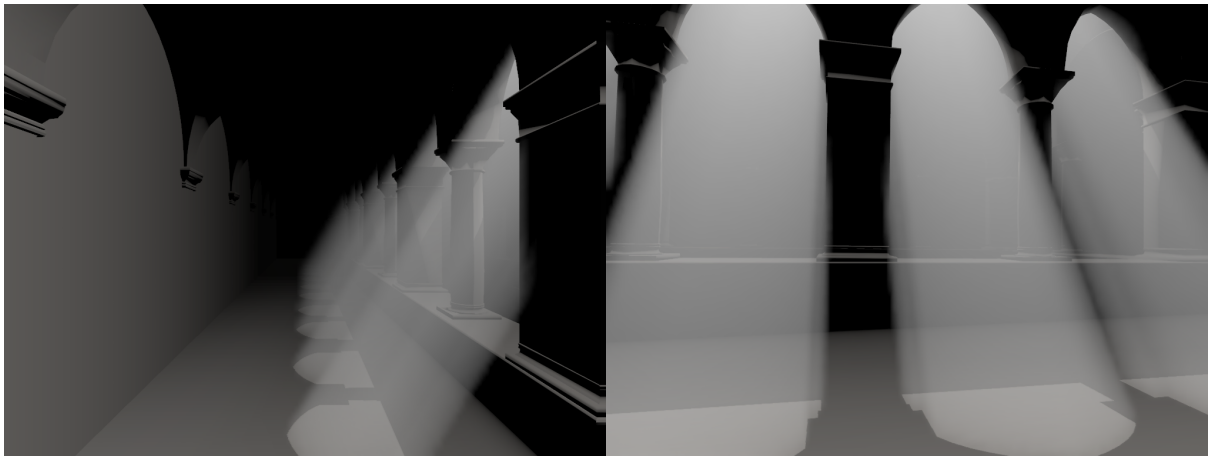
Volumetric Fog



목차

Volumetric Fog란

Volumetric Fog는 물리 기반의 조명 산란을 계산하여 입체적인 안개를 표현하는 기법입니다. 안개 외에도 빌보드나 포스트 프로세스를 대체하여 갓 레이를 표현하는 데도 사용할 수 있습니다.

Volumetric Fog는 SIGGRAPH 2014에서 Ubisoft가 발표한 “Volumetric fog: Unified, compute shader based solution to atmospheric scattering” 을 통해서 소개되었습니다. 이 글에서는 해당 자료를 기반으로 Direct3D 11/12로 구현한 결과물을 통해 Volumetric Fog의 기본적인 구현 방법을 소개하고 언리얼 엔진에서 Volumetric Fog가 어떻게 구현되어 있는지 살펴보도록 하겠습니다.

구현 방법

Volumetric Fog는 다음과 같은 순서로 구현됩니다.

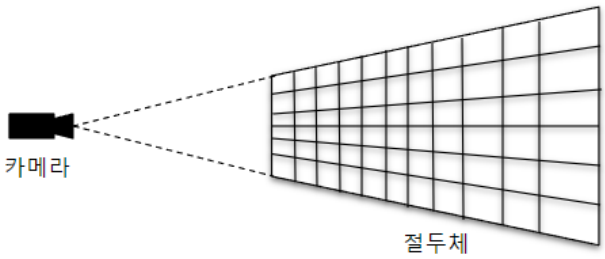
1. 절두체에 대한 볼륨 텍스쳐를 생성
2. 컴퓨트 셰이더를 통해 볼륨 텍스쳐의 각 텍셀에 대한 산란 및 밀도 계산

3. 카메라부터의 거리에 따른 산란 누적
4. Scene에 Fog적용

이후로 각 단계를 좀 더 자세히 알아보도록 하겠습니다.

볼륨 텍스쳐 생성

가장 먼저 볼륨 텍스쳐를 생성할 필요가 있습니다. 볼륨 텍스쳐는 3D공간의 산란 및 밀도 정보를 저장하는 데 사용됩니다.



발표 자료에 따르면 볼륨 텍스쳐의 크기는 720p기준으로 플랫폼에 따라 160x90x64(가로x세로x깊이) 혹은 160x90x128이며 포맷으로는 16bit RGBA Float를 사용하였다고 합니다.

저는 160x90x128 크기의 볼륨 텍스쳐를 UAV와 SRV로 사용할 수 있도록 하여 생성하였습니다. 볼륨 텍스쳐 생성을 위한 설정은 다음과 같습니다.

```
const std::array<uint32, 3>& frustumGridSize = Proxy()->FrustumGridSize();

agl::TextureTrait frustumVolumeTrait = {
  .m_width = frustumGridSize[0], // 160
  .m_height = frustumGridSize[1], // 90
  .m_depth = frustumGridSize[2], // 128
  .m_sampleCount = 1,
  .m_sampleQuality = 0,
  .m_mipLevels = 1,
  .m_format = agl::ResourceFormat::R16G16B16A16_FLOAT,
  .m_access = agl::ResourceAccessFlag::GpuRead | agl::ResourceAccessFlag::GpuWrite,
  .m_bindType = agl::ResourceBindType::RandomAccess | agl::ResourceBindType::ShaderResource,
  .m_miscFlag = agl::ResourceMisc::Texture3D
};
```

이렇게 생성한 볼륨 텍스쳐의 각 텍셀은 카메라 공간의 일정 복셀 영역에 대한 산란 및 밀도 정보를 저장하는데 텍스쳐의 크기에 제약이 있기 때문에 카메라 공간을 균등하게 분할하는 것이 아니라 가까운 곳에 더 높은 해상도를 제공하기 위해 z축이(카메라가 보는 방향) 지수 분포를 따르도록 분할 합니다. 볼륨 텍스쳐 텍셀의 id를 받아 월드 공간 좌표로 변환하는 함수를 통해 자세한 공간 분할 과정을 알아보겠습니다.

```
/*
- input
id : 볼륨 텍스쳐 텍셀의 id (x,y,z)
dims : 볼륨 텍스쳐 크기 (width, height, depth)

- output
월드 공간 좌표
*/
float3 ConvertThreadIdToWorldPosition( uint3 id, uint3 dims )
{
  // id -> ndc
  float3 ndc = ConvertThreadIdToNdc( id, dims );
  float depth = ConvertNdcZToDepth( ndc.z );

  return ConvertToWorldPosition( ndc, depth );
}
```

먼저 볼륨 텍스쳐의 텍셀 id와 텍스쳐 크기를 인자로 받아 NDC(Normalized Device Coordinate) 공간으로 변경합니다. 이 과정을 담당하는 변환 함수 ConvertThreadIdToNdc는 다음과 같습니다.

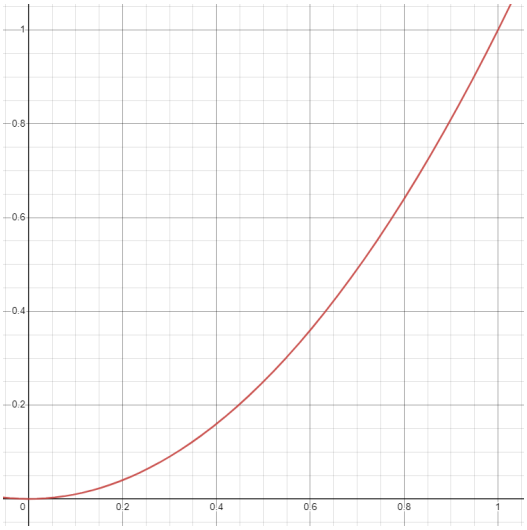
```
/*
- input
```

```
id : 볼륨 텍스쳐 텍셀의 id (x,y,z)
dims : 볼륨 텍스쳐 크기 (width, height, depth)

- output
Direct3D 왼손 좌표계를 따르므로 X,Y 가 -1 ~ 1 Z가 0 ~ 1 범위인 NDC 공간의 좌표
*/
float3 ConvertThreadIdToNdc( uint3 id, uint3 dims )
{
  float3 ndc = id;
  ndc += 0.5f;
  ndc *= float3( 2.f / dims.x, -2.f / dims.y, 1.f / dims.z );
  ndc += float3( -1.f, 1.f, 0.f );
  return ndc;
}
```

NDC 공간의 좌표를 얻고 나면 Z좌표 값을 통해 지수 분포를 따라는 카메라 공간 깊이 값을 얻습니다. 이 과정은 ConvertNdcZToDepth함수에서 이뤄집니다.

```
/*
- input
ndcZ : NDC 공간의 Z좌표

- global parameter
VolumetricFogParam.DepthPackExponent : 지수 분포 변환을 위한 지수
VolumetricFogParam.NearPlaneDist : Fog에 대한 근평면까지의 거리
VolumetricFogParam.FarPlaneDist : Fog에 대한 원평면까지의 거리

- ouput
지수 분포를 따라는 카메라 공간의 깊이
*/
float ConvertNdcZToDepth( float ndcZ )
{
  float depthPackExponent = VolumetricFogParam.DepthPackExponent;
  float nearPlaneDist = VolumetricFogParam.NearPlaneDist;
  float farPlaneDist = VolumetricFogParam.FarPlaneDist;

  return pow( ndcZ, depthPackExponent ) * ( farPlaneDist - nearPlaneDist ) + nearPlaneDist;
}
```

pow를 통해 ndcZ를 지수 분포로 변환합니다. 지수는 스크립트를 통해 조절할 수 있지만 기본적으로 2를 사용하고 있습니다. 이 결과 ndcZ의 분포는 다음과 같이 변하게 됩니다.



가로축을 ndcZ라고 할 때 낮은 값에서는 세로축의 수치 변화가 완만하고 높은 값에서는 급격해지는 것을 볼 수 있습니다. 이를 통해 카메라에 가까운 위치에 좀 더 높은 해상도를 제공하게 됩니다.

이제 앞에서 얻은 NDC 공간 좌표와 카메라 공간의 깊이 값을 통해 월드 위치를 계산할 수 있습니다.

```
/*
- input
ndc : NDC 공간 좌표
depth : 카메라 공간의 깊이 값
```

```
- global parameter
InvProjectionMatrix : 투영변환의 역행렬
InvViewMatrix : 카메라 변환의 역행렬

- output
월드 공간 좌표
*/
float3 ConvertToWorldPosition( float3 ndc, float depth )
{
  // view ray
  // ndc좌표에 대한 카메라 공간 광선을 계산
  float4 viewRay = mul( float4( ndc, 1.f ), InvProjectionMatrix );
  viewRay /= viewRay.w;
  viewRay /= viewRay.z; // z값이 1이 되도록

  // ndc -> world position
  float4 worldPosition = mul( float4( viewRay.xyz * depth, 1.f ), InvViewMatrix );

  return worldPosition.xyz;
}
```

산란 및 밀도 계산

산란 계산

산란은 파동이 입자와 충돌하여 주변으로 흩어지는 현상입니다. 빛이 진공이 아닌 어떤 매질을 통과하면 매질의 입자와 충돌하여 여러 방향으로 분산되거나 흡수됩니다. 이 단계에서는 볼륨 텍스쳐의 각 위치에서 카메라까지 얼마나 많은 양의 빛이 산란하는지를 계산하고 산란 누적 단계에서 사용될 매질의 밀도를 기록합니다.

Phase function

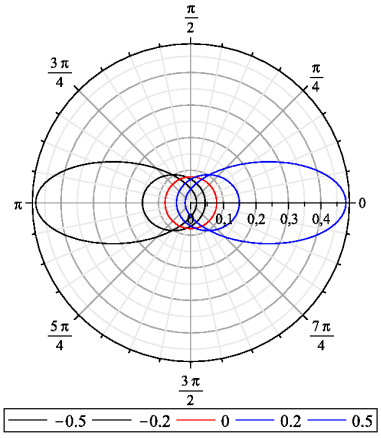
먼저 빛이 얼마나 산란하는지 계산하기 위해서 phase function을 사용합니다.

phase function은 빛이 모든 방향으로 얼마나 산란하는지를 나타내는 함수입니다. 이 함수는 조명 벡터와 나가는 방향 벡터 사이의 각도에 대한 함수인데 에너지 보존 법칙을 따르기 때문에 모든 방향에 대해 적분하면 1이 됩니다. (phase function에 흡수가 고려되었다면 1보다 작을 수 있습니다.)

phase function은 산란의 종류에 따라 몇 가지가 있는데 Volumetric Fog 구현에서는 Henyey-Greenstein phase function 함수를 사용합니다.

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

Henyey-Greenstein phase function



이방성 계수 g와 각도에 따른 그래프

Henyey-Greenstein phase function은 미(Mie) 산란과 같은 이방성 산란을 시뮬레이션하는 가장 일반적인 phase function입니다.

미 산란은 빛의 파장과 크기가 비슷한 입자와 충돌하여 발생하는 산란으로 구름이나 비가 오기 전 하늘이 뿌옇게 보이는 것은 미 산란에 의한 현상입니다.

```
/*
- input
wi : 빛이 들어오는 방향
wo : 빛이 나가는 방향
```

```
g : 이방성 계수

- output
wi에서 들어와 wo로 나가는 빛의 양
*/
float HenyeyGreensteinPhaseFunction( float3 wi, float3 wo, float g )
{
  float cosTheta = dot( wi, wo );
  float g2 = g * g;
  float denom = pow( 1.f + g2 - 2.f * g * cosTheta, 3.f / 2.f );
  return ( 1.f / ( 4.f * PI ) ) * ( ( 1.f - g2 ) / max( denom, EPSILON ) );
}
```

HenyeyGreensteinPhaseFunction을 이용해 볼륨 텍스쳐의 각 텍셀에서 카메라 방향으로 산란하는 빛의 양을 계산할 수 있습니다.

밀도 계산

펄린 노이즈 텍스쳐를 이용하거나 높이에 따라 지수적으로 분포하는 함수를 사용하여 매질의 불균등한 밀도를 표현할 수 있지만 제 결과물에서는 매질의 밀도가 균등한 경우만을 구현하였습니다. 따라서 볼륨 텍스쳐에 기록될 밀도는 다음과 같습니다.

```
/*
UniformDensity : 균등한 밀도
*/
float density = UniformDensity;
```

산란 계산과 밀도 계산을 합친 컴퓨트 셰이더 코드는 다음과 같습니다.

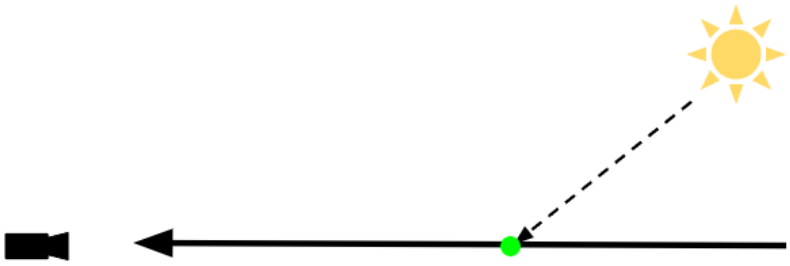
산란한 빛의 색상은 볼륨 텍스쳐의 rgb채널에 밀도는 a채널에 저장합니다.

```
/*
- global parameter
FrustumVolume : 볼륨 텍스쳐
CameraPos : 카메라 위치
UniformDensity : 균등한 밀도
HemisphereUpperColor : Ambient로 사용하는 반구 조명 색상
Intensity : 빛 강도 조절을 위한 변수

*/
[numthreads( 8, 8, 8 )]
void main( uint3 DTid : SV_DispatchThreadId )
{
  uint3 dims;
  FrustumVolume.GetDimensions( dims.x, dims.y, dims.z );

  if ( all( DTid < dims ) )
  {
    float3 worldPosition = ConvertThreadIdToWorldPosition( DTid, dims );
    float3 toCamera = normalize( CameraPos - worldPosition );

    float density = UniformDensity;

    float3 lighting = HemisphereUpperColor.rgb * HemisphereUpperColor.a;
    [loop]
    for ( uint i = 0; i < NumLights; ++i )
    {
      ForwardLightData light = GetLight( i );

      float3 lightDirection = { 0.f, 0.f, 0.f };

      if ( length( light.m_direction ) > 0.f ) // Directional Light의 경우
      {
        lightDirection = normalize( light.m_direction );
      }
      else // 그 외의 경우 (Point, Spot)
      {
        lightDirection = normalize( worldPosition - light.m_position );
      }

      float3 toLight = -lightDirection;
      float visibility = Visibility( worldPosition, toLight ); // 섀도우 맵으로 가시성 검사
      float phaseFunction = HenyeyGreensteinPhaseFunction( lightDirection, toCamera, AsymmetryParameterG );
      lighting += visibility * light.m_diffuse.rgb * light.m_diffuse.a * phaseFunction;
    }

    FrustumVolume[DTid] = float4( lighting * Intensity * density, density )
```

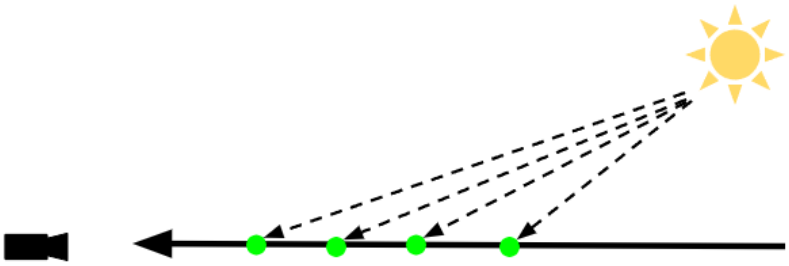
```
  }
}
```

산란 누적

앞선 단계에서는 볼륨 텍스쳐의 각 텍셀 위치에서 카메라 방향으로 산란하는 빛의 양과 밀도를 계산하였습니다. 지금까지 볼륨 텍스쳐에 저장된 정보는 아직 3차원 공간의 한 점에서의 산란에 지나지 않습니다.



산란 누적 단계에서는 텍셀 위치에서 카메라까지 빛이 이동하는 경로의 산란을 누적하고 빛이 매질을 통과하면서 발생하는 빛의 감쇠 현상을 시뮬레이션합니다.



Beer-Lambert 법칙

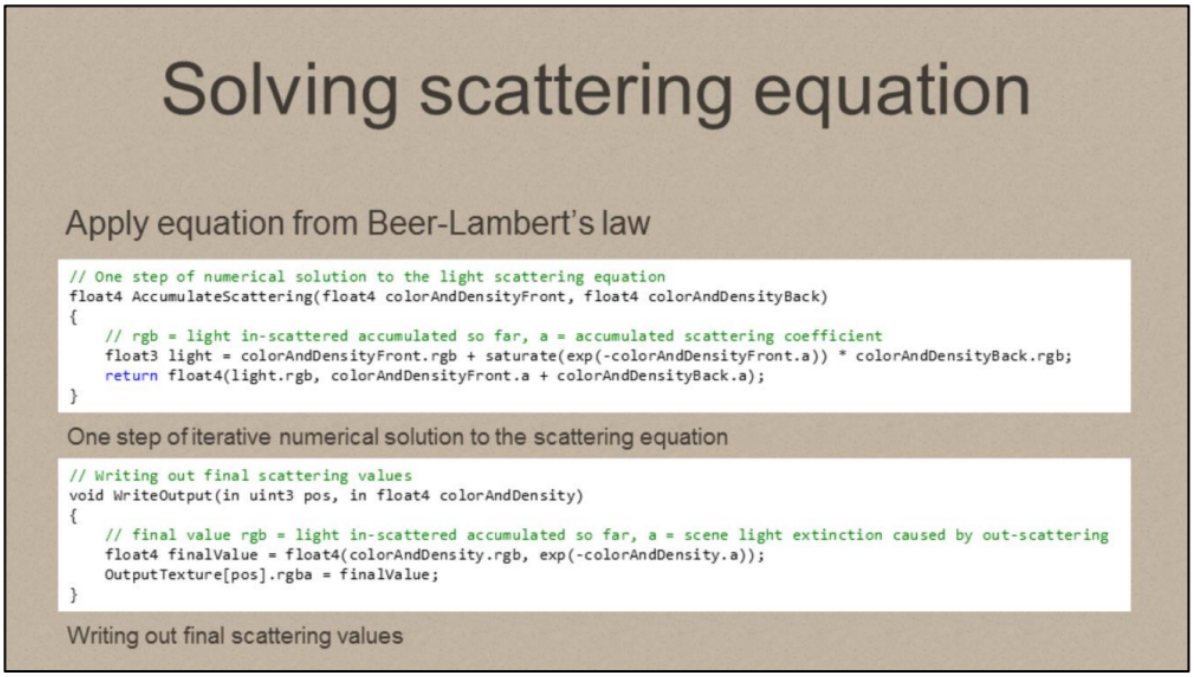
Beer-Lambert 법칙은 매질의 성질과 빛의 감쇠현상에 대한 법칙입니다. 이 법칙은 투과율을 정의하는데 투과율은 주어진 방향에서 들어오는 빛이 매질을 통해 전달되는 비율로 A위치에서 B까지의 투과율은 일반적으로 다음과 같이 정의됩니다.

$$T(A \rightarrow B) = e^{-\int_B^A \beta_e(x)dx}$$

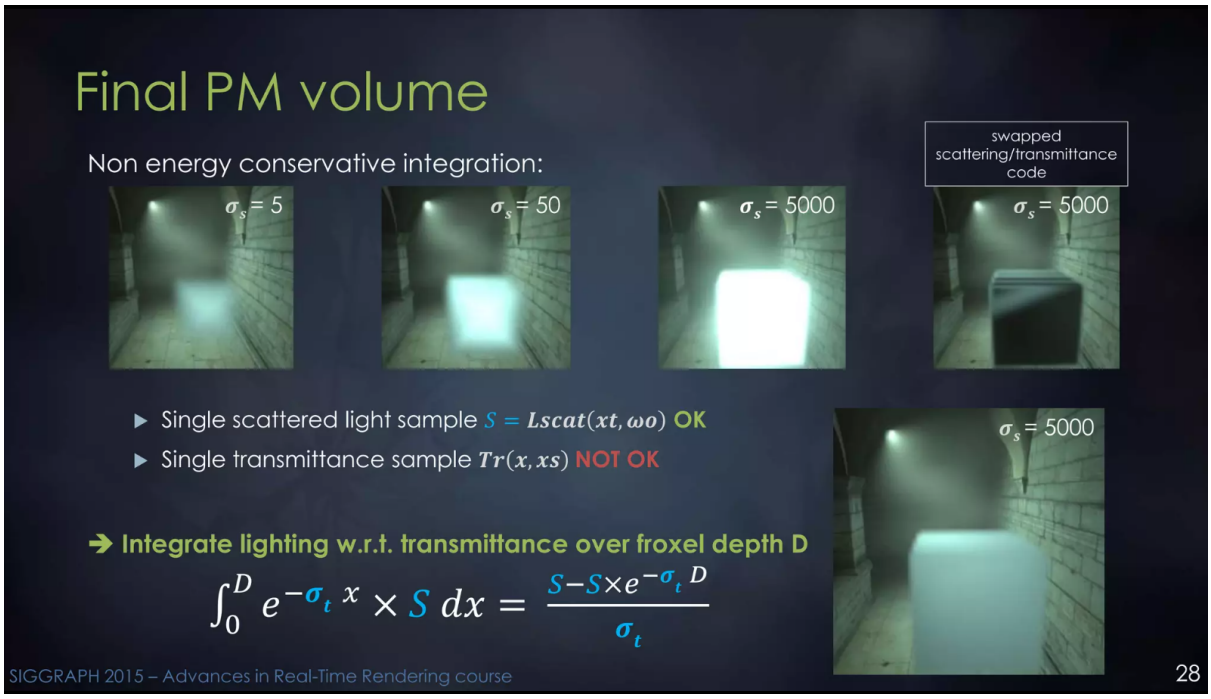
여기서 β_e는 산란과 흡수의 합으로 정의되는 소멸 계수입니다. 이 구현에서는 밀도가 β_e가 됩니다.

또한 투과율을 그래프로 그려보면 아래와 같이 거리에 따라 지수적으로 감소하는 것을 확인할 수 있습니다.

Beer-Lambert 법칙을 적용하여 Ubisoft의 발표 자료에서는 다음과 같은 누적 코드를 제시하고 있습니다.



그러나 SIGGRAPH 2015에서 Frostbite가 발표한 "Physically Based and Unified Volumetric Rendering in Frostbite"에 따르면 이러한 누적 방식은 에너지 보존 법칙을 준수하지 않고 있어 새로운 누적 방식을 제시하고 있습니다.

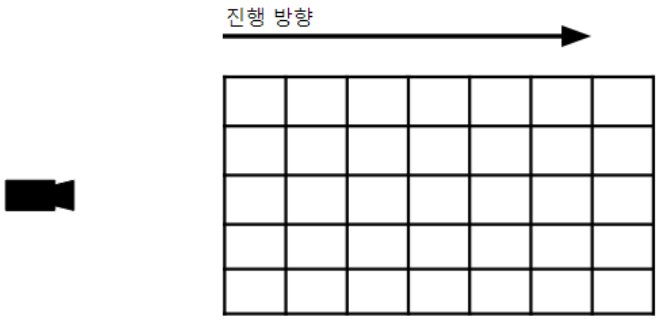


산란 계수 σ_s가 커짐에 따라 점점 밝아지는 것을 확인할 수 있습니다. Frostbite의 누적식의 각 기호는 다음을 의미하니 참고하시기 바랍니다.

- D : 볼륨 텍스쳐 텍셀까지의 깊이 (= 매질 안 빛의 이동 거리)
- σ_t : 산란과 흡수를 합친 소멸 계수 (= 구현 코드에서는 밀도)
- S : 산란한 조명

앞으로 볼 구현 코드에서는 Frostbite의 누적 방식을 사용하였습니다.

누적 방향은 카메라 위치에서 가까운 곳에서 먼 곳으로 이동하며 누적합니다. 즉 볼륨 텍스쳐의 깊이 0에서 부터 시작하게 됩니다.



이에 따른 컴퓨트 셰이더 코드는 다음과 같습니다.

```
RWTexture3D<float4> FrustumVolume : register( t0 );

[numthreads( 8, 8, 1 )]
void main( uint3 DTid : SV_DispatchThreadID )
{
  uint3 dims;
  FrustumVolume .GetDimensions( dims.x, dims.y, dims.z );

  if ( all( DTid < dims ) )
  {
    float4 accum = float4( 0.f, 0.f, 0.f, 1.f );
    uint3 pos = uint3( DTid.xy, 0 );

    [loop]
    for ( uint z = 0; z < dims.z; ++z )
    {
      pos.z = z;
      float4 slice = FrustumVolume[pos];
      float tickness = SliceTickness( (float)z / dims.z, dims.z );

      accum = ScatterStep( accum.rgb, accum.a, slice.rgb, slice.a, tickness );
      FrustumVolume [pos] = accum;
    }
  }
}
```

SliceTickness를 통해서 볼륨 텍스쳐의 현재 위치와 한 칸 앞의 차이를 통해 지수 분포가 적용된 깊이 두께를 계산합니다. 이 두께가 D 입니다.

```
/*
- input
ndc : NDC 공간 좌표
dimZ : 볼륨 텍스쳐의 깊이

- output
깊이 두께
*/
float SliceTickness( float ndcZ, uint dimZ )
{
  return ConvertNdcZToDepth( ndcZ + 1.f / float( dimZ ) ) - ConvertNdcZToDepth( ndcZ );
}
```

실제 누적이 이뤄지는 ScatterStep 함수는 다음과 같습니다.

```
/*
- input
accumulatedLight : 누적된 조명
accumulatedTransmittance : 누적된 투과율
sliceLight : 현재 위치의 조명
sliceDensity : 현재 위치의 밀도
tickness : 깊이 두께

- global constant
DensityScale : 밀도에 대한 스케일(균일한 밀도 파라미터를 소수점이 아닌 좀 더 큰 값으로 사용하기 위해 적절히 조정한 수치)

- output
누적된 빛(rgb)과 투과율(a)
*/
```

```
static const float DensityScale = 0.01f;

float4 ScatterStep( float3 accumulatedLight, float accumulatedTransmittance, float3 sliceLight, float sliceDensity, float tickness )
{
  sliceDensity = max( sliceDensity, 0.000001f );
  sliceDensity *= DensityScale;
  float sliceTransmittance = exp( -sliceDensity * tickness );

  // Frostbite의 누적 방식
  float3 sliceLightIntegral = sliceLight * ( 1.f - sliceTransmittance ) / sliceDensity;

  accumulatedLight += sliceLightIntegral * accumulatedTransmittance;
  accumulatedTransmittance *= sliceTransmittance;

  return float4( accumulatedLight, accumulatedTransmittance );
}
```

Fog 적용

이제 준비 과정은 모두 끝났습니다. 적용 단계에서는 화면 해상도만큼의 사각형을 그려 장면에 Fog를 적용합니다. 이 과정에서 사용하는 주요 렌더 스테이트는 다음과 같습니다.

```
// SrcAlpha로 블렌드
BlendOption volumetricFogDrawPassBlendOption;
RenderTargetBlendOption& rt0BlendOption = volumetricFogDrawPassBlendOption.m_renderTarget[0];
rt0BlendOption.m_blendEnable = true;
rt0BlendOption.m_srcBlend = agl::Blend::One;
rt0BlendOption.m_destBlend = agl::Blend::SrcAlpha;
rt0BlendOption.m_srcBlendAlpha = agl::Blend::Zero;
rt0BlendOption.m_destBlendAlpha = agl::Blend::One;

// 깊이 테스트 및 쓰기 OFF
DepthStencilOption depthStencilOption;
depthStencilOption.m_depth.m_enable = false;
depthStencilOption.m_depth.m_writeDepth = false;
```

픽셀 셰이더는 각 픽셀에 대한 카메라 공간 좌표를 계산해서 해당 좌표를 볼륨 텍스쳐의 UV로 변경한 뒤 샘플링합니다. 블렌드 스테이트 SrcAlpha로 블렌드하도록 설정하였기 때문에 투과율이 담긴 볼륨 텍스쳐의 알파 채널을 그대로 사용하면 투과율이 1에 가까울수록 씬의 색상이 투과율이 0에 가까울수록 산란한 빛의 색상이 반영됩니다.

```
float4 main( PS_INPUT input ) : SV_Target0
{
  float viewSpaceDistance = ViewSpaceDistance.Sample( ViewSpaceDistanceSampler, input.uv ).x;
  if ( viewSpaceDistance <= 0.f )
  {
    viewSpaceDistance = FarPlaneDist;
  }

  float3 viewPosition = normalize( input.viewRay ) * viewSpaceDistance;

  float3 uv = float3( input.uv, ConvertDepthToNdcZ( viewPosition.z ) );

  float4 scatteringColorAndTransmittance = AccumulatedVolume.Sample( AccumulatedVolumeSampler, uv );
  float3 scatteringColor = HDR( scatteringColorAndTransmittance.rgb );

  return float4( scatteringColor, scatteringColorAndTransmittance.a );
}
```

품질 향상을 위한 개선점

여기까지 기본적인 Volumetric Fog의 구현 방법을 살펴보았습니다. 지금부터는 품질 향상을 위한 개선점 2가지를 소개하고 적용 전과 후를 비교해 보겠습니다.

1) Temporal reprojection

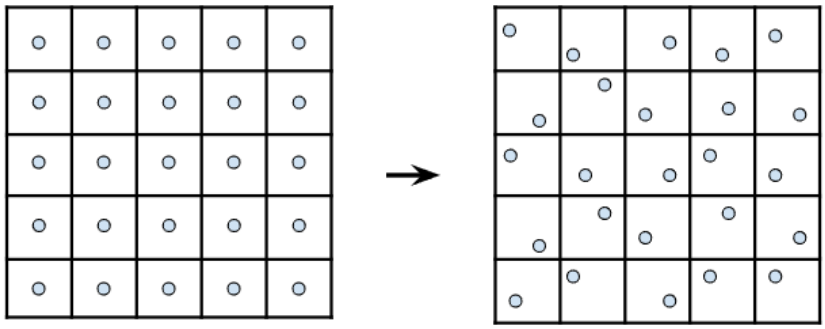
Ubisoft의 발표 자료에서 제시한 볼륨 텍스쳐의 크기는 720p 기준 160x90x128입니다. 볼륨 텍스쳐의 해상도가 화면의 해상도 보다 작기 때문에 필연적으로 볼륨 텍스쳐의 텍셀 하나가 여러 픽셀을 커버하게 됩니다.

720p의 가로세로(1280 x 720)를 볼륨 텍스쳐의 텍셀 수로 나눠보면 가로세로 8픽셀씩 총 64픽셀을 커버하고 있습니다. 64픽셀이 모두 동일한 산란 값을 사용해야 하므로 Volumetric Fog는 언더 샘플링으로 인한 아티팩트가 발생합니다.

Temporal reprojection은 이런 아티팩트를 제거하기 위해 사용됩니다. Temporal 이라는 단어에서 알 수 있듯이 Temporal Anti-Aliasing과 마찬가지로 이전 프레임에 계산했던 결과를 누적하여 언더 샘플링을 해결하려는 방식입니다. 따라서 이전 프레임의 정보를 유지하기 위해 다수의 텍스쳐가 필요합니다.

```
// 이전 프레임, 현재 프레임용 텍스쳐 2개, 스왑하면서 사용
for ( agl::RefHandle<agl::Texture>& frustumVolume : m_frustumVolume )
{
  frustumVolume = agl::Texture::Create( frustumVolumeTrait );
  EnqueueRenderTask(
    [texture = frustumVolume]()
    {
      texture->Init();
    } );
}

// 산란 누적 단계에서 사용할 결과 텍스쳐 1개
m_accumulatedVolume = agl::Texture::Create( frustumVolumeTrait );
EnqueueRenderTask(
  [texture = m_accumulatedVolume]()
  {
    texture->Init();
  } );
```

Temporal Anti-Aliasing이 매 프레임 jitter를 통해 렌더링되는 물체의 위치를 조금씩 어긋나게 했던 것처럼 Temporal reprojection도 jitter를 통해 볼륨 텍스쳐 텍셀의 월드 위치를 조금씩 어긋나게 합니다.



여기에는 Halton Sequence를 사용하였습니다. (Halton Sequence의 설명은 이전 Temporal Anti-Aliasing 페이지를 참고 부탁드립니다.) 3차원 공간에 대한 jitter를 위해 Halton Sequence도 3차원으로 확장하였습니다.

```
// 2, 3, 5(소수)로 뽑은 수열
static const float3 HALTON_SEQUENCE[MAX_HALTON_SEQUENCE] = {
  float3( 0.5, 0.333333, 0.2 ),
  float3( 0.25, 0.666667, 0.4 ),
  float3( 0.75, 0.111111, 0.6 ),
  float3( 0.125, 0.444444, 0.8 ),
  float3( 0.625, 0.777778, 0.04 ),
  float3( 0.375, 0.222222, 0.24 ),
  float3( 0.875, 0.555556, 0.44 ) ,
  float3( 0.0625, 0.888889, 0.64 ),
  float3( 0.5625, 0.037037, 0.84 ),
  float3( 0.3125, 0.37037, 0.08 ),
  float3( 0.8125, 0.703704, 0.28 ),
  float3( 0.1875, 0.148148, 0.48 ),
  float3( 0.6875, 0.481482, 0.68 ),
  float3( 0.4375, 0.814815, 0.88 ),
  float3( 0.9375, 0.259259, 0.12 ),
  float3( 0.03125, 0.592593, 0.32 )
};
```

산란 및 밀도 계산 단계에서 다음과 같이 jitter를 추가합니다.

```
float3 jitter = HALTON_SEQUENCE[( FrameCount + DTid.x + DTid.y * 2 ) % MAX_HALTON_SEQUENCE];
jitter.xyz -= 0.5f; // -0.5 ~ 0.5 범위로 조정

// jitter가 적용된 월드 위치
float3 worldPosition = ConvertThreadIdToWorldPosition( DTid, dims, jitter );
```

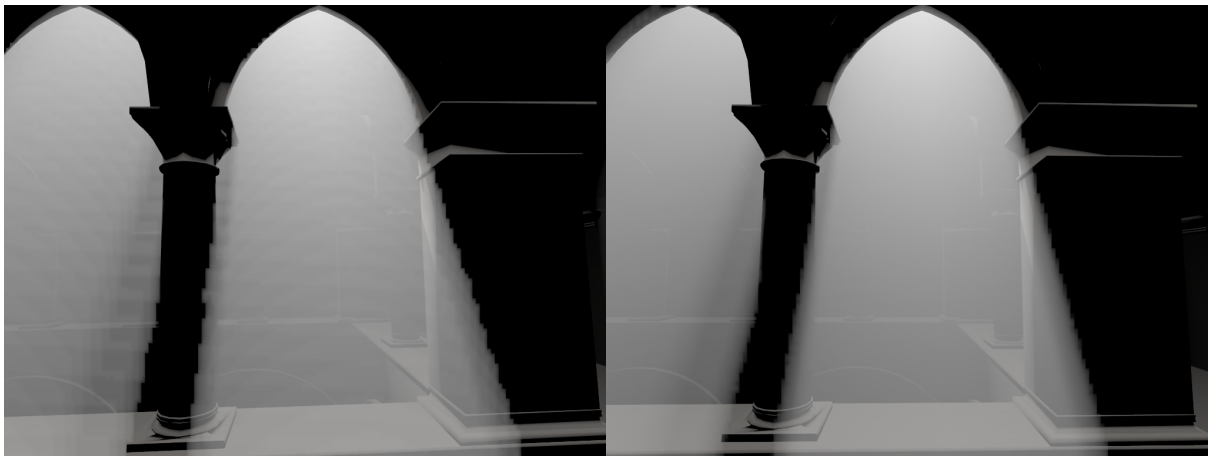
그리고 이전 프레임의 정보가 현재도 유효하다면 (=카메라 밖으로 나가지 않았다면) 현재 프레임과 섞습니다.

```
/*
- Global Parameter
TemporalAccum : Temporal reprojection 사용 여부, 첫 프레임에 꺼야할 때 사용 ( 이전 프레임 정보가 없기 때문 )
PrevViewProjectionMatrix : 이전 프레임 카메라 투영 행렬
*/
Texture3D HistoryVolume : register( t0 );
SamplerState HistorySampler : register( s0 );

// ...

// Main 함수 내부 현재 프레임 산란 계산 후
curScattering = float4( lighting * Intensity * density, density );

if ( TemporalAccum > 0.f )
{
  float3 worldPosWithoutJitter = ConvertThreadIdToWorldPosition( DTid, dims );
  // 이전 프레임의 UV 계산
  float3 prevFrameUV = ConvertWorldPositionToUV( worldPosWithoutJitter, PrevViewProjectionMatrix );

  // 0 ~ 1사이 범위면 유효
  if ( all( prevFrameUV <= ( float3 )1.f ) && all( prevFrameUV >= ( float3 )0.f ) )
  {
    float4 prevScattering = HistoryVolume.SampleLevel( HistorySampler, prevFrameUV, 0 );
    curScattering = lerp( prevScattering, curScattering, 0.05f );
  }
}

FrustumVolume[DTid] = curScattering;
```

산란 누적 단계는 큰 변화는 없고 별도의 볼륨 텍스쳐에 출력하도록 수정되었습니다.

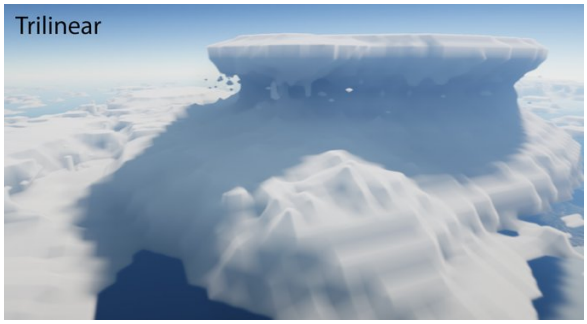
```
AccumulatedVolume[pos] = accum;
```

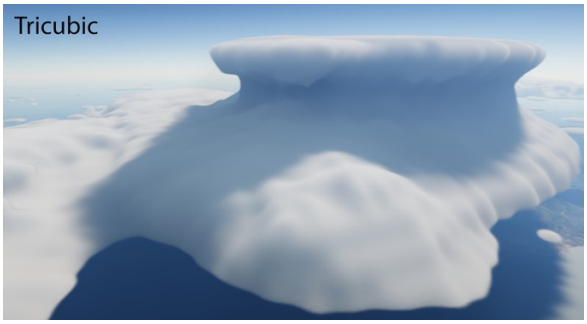
적용 전후를 비교해 보면 다음과 같습니다.



2) Tricubic texture sampling

복셀 데이터에 대한 샘플링 방식을 변경하는 것으로 추가적인 품질 향상을 얻을 수 있습니다. Tricubic texture sampling은 “CUDA Cubic B-Spline Interpolation”에 따른 방식으로 볼륨 텍스쳐를 B-Spline을 통해 보간합니다. 선형 샘플링보다 더 나은 결과를 관찰할 수 있습니다.



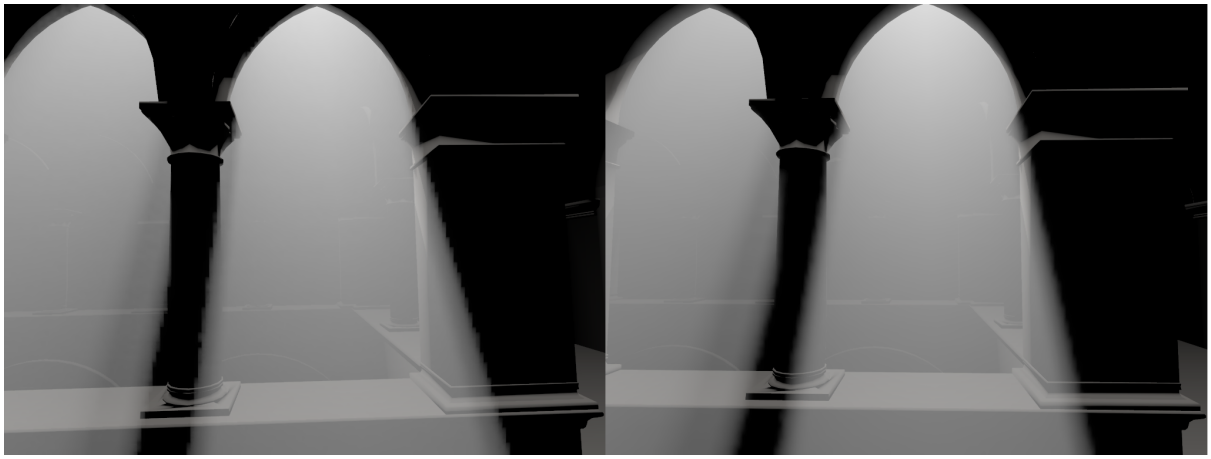


출처 : https://twitter.com/FewesW/status/1300045087938879489

구현에는 유니티용 Tricubic sampling 코드를 개인 프로젝트 상황에 맞게 약간 수정하여 Fog 적용 단계에서 사용하였습니다.

```
#if TricubicTextureSampling == 1
  float4 scatteringColorAndTransmittance = Tex3DTricubic( AccumulatedVolume, AccumulatedVolumeSampler, uv);
#else
  float4 scatteringColorAndTransmittance = AccumulatedVolume.Sample( AccumulatedVolumeSampler, uv );
#endif
```

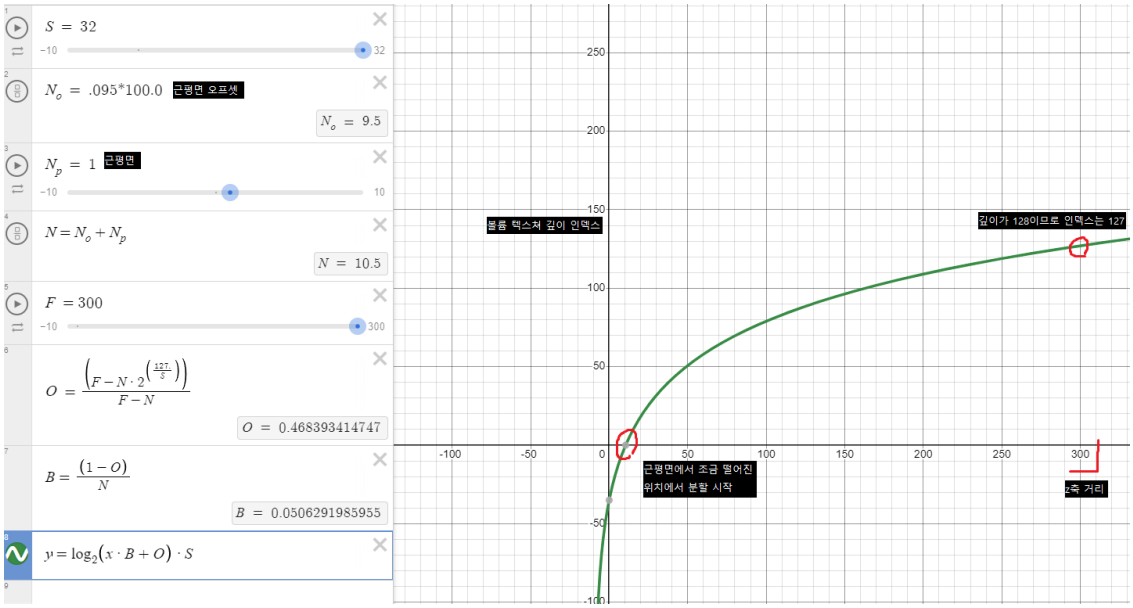
적용 전후를 비교해 보면 다음과 같습니다.



마치며

언리얼 엔진의 Volumetric Fog구현도 여기 다룬 내용과 크게 다르지 않습니다. 몇 가지 개선점 (Temporal reprojection 실패 시 루프를 돌아 프레임을 누적하여 빠르게 수렴할 수 있도록 하는 등)이 있지만 소개해 드린 내용을 바탕으로 충분히 이해할 수 있을 정도라고 생각합니다. 언리얼의 구현도 추가로 살펴보고 싶으신 분을 위해 도움이 될 수 있는 진입점 몇 곳을 여기에 남기도록 하겠습니다.

- VolumetricFog.cpp : 컴퓨트 셰이더를 사용하는 Volumetric Fog 산란 계산 코드들이 모여있음.
  - ComputeVolumetricFog() : 가장 처음 진입점 함수입니다.
  - GetVolumetricFogGridSize() : 볼륨 텍스쳐 크기 계산 함수입니다.
  - GetVolumetricFogGridZParams() : Z축 분할에 사용되는 파라미터 계산 함수, 언리얼에서는 다음과 같은 분포를 따르도록 Z축을 분할하는데 근평면에 너무 많은 해상도가 할당되는 것을 방지하기 위함으로 생각합니다.



- VoxelizeFogVolumePrimitives() : Volume 머티리얼을 통해 렌더링 하는 프리미티브를 복셀화 해서 Volumetric Fog 볼륨 텍스쳐에 렌더링 하는 함수입니다. 이를 통해 다양한 밀도를 표현할 수 있습니다. 상자와 구 두가지 Voxelize 모드가 있습니다.
- FVolumetricFogLightScatteringCS : 산란 계산 컴퓨트 셰이더 클래스입니다.
- FVolumetricFogFinalIntegrationCS : 산란 누적 컴퓨트 셰이더 클래스입니다.
- RenderViewFog() : Fog를 렌더링하는 함수입니다.
- FExponentialHeightFogPS : Fog를 렌더링하는 픽셀 셰이더 클래스입니다.

준비한 내용은 여기까지입니다. 제 개인 프로젝트 구현의 전체 코드는 아래 github를 참고 부탁드립니다.

GitHub - xtozero/SSR at volumetric_fog
Screen Space Reflection. Contribute to xtozero/SSR development by creating an account on GitHub.
https://github.com/xtozero/SSR/tree/volumetric_fog
xtozero/SSR
Screen Space Reflection
1 Contributor 1 Issue 13 Stars 1 Fork

감사합니다.

Reference

- https://github.com/diharaw/volumetric-fog : opengl 구현
- https://github.com/bartwronski/CSharpRenderer : 원 저자의 구현
- https://github.com/Unity-Technologies/VolumetricLighting/tree/master/Assets/VolumetricFog : Unity의 구현