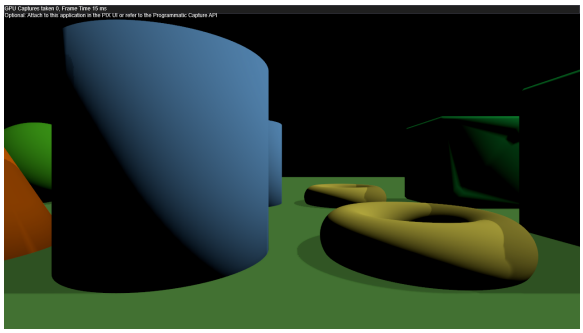
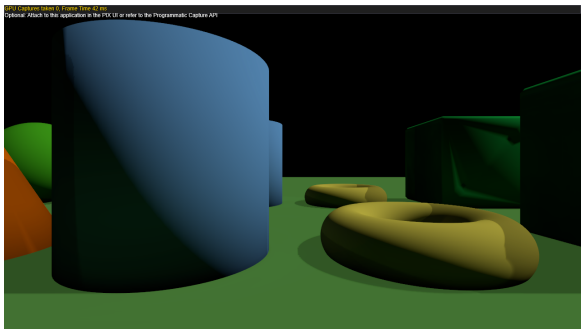
Reflective Shadow Maps



RSMs Off



RSMs On

목차

Reflective Shadow Maps 란

Reflective Shadow Maps(이하 RSMs)은 Shadow Maps라는 이름에서 유추되는 것과는 다르게 그림자를 그리는 데 사용되는 기법은 아닙니다. RSMs는 그럴듯한 간접광(Indirect Light)을 근사하는 방식으로 실시간으로 전역 조명(Global Illumination)을 표현하는 알고리즘입니다. 다만 그 방식이 표준 섀도우 맵(Standard Shadow Map, 이하 SSM)을 확장하는 방식이기 때문에 이러한 이름이 붙었다고 생각됩니다.

이 글에서는 RSMs를 소개한 논문(Dachsbacher04)을 기반으로 Direct3D 11/12로 구현한 결과물을 통해 RSMs의 기본적인 구현 방법을 소개하겠습니다.

구현 방식

RSMs는 다음과 같은 순서로 구현됩니다.

1. SSM을 확장하여 다음과 같은 정보를 텍스쳐에 기록합니다.
    a. 월드 좌표 위치 (World Position)
    b. 월드 노멀 (World Normal)

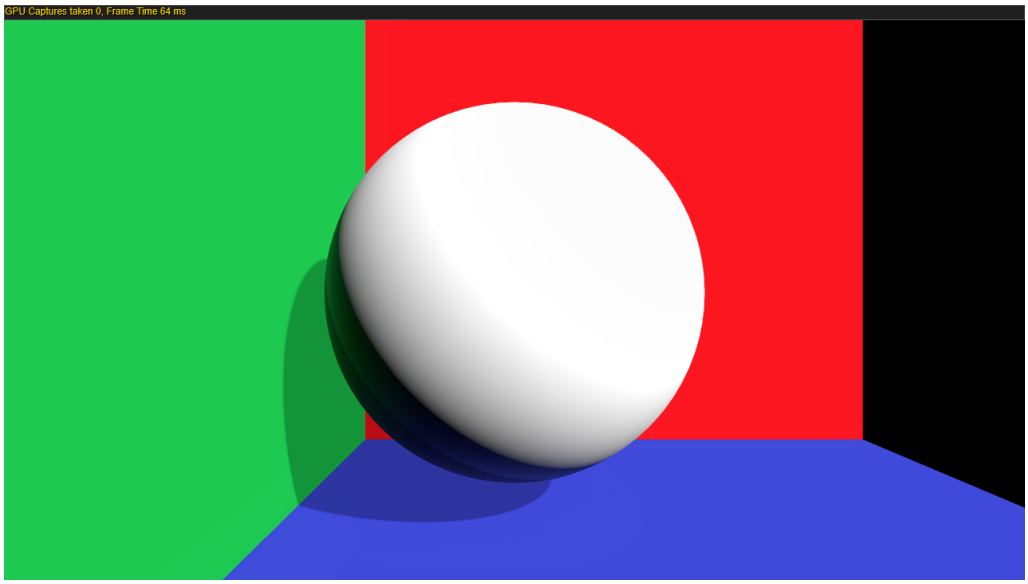
c. 반사된 방사속 (Reflected Radiant Flux)

2. 전체 화면 크기의 사각형을 그리면서 이전 단계에서 텍스쳐에 기록한 정보를 바탕으로 간접광을 계산합니다.
3. 직접광을 계산하면서 간접광 계산 결과를 합성하여 최종 결과를 계산합니다.

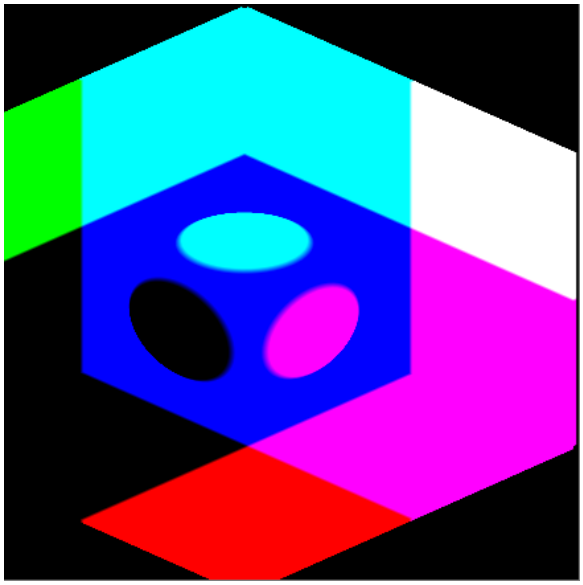
이후로 각 단계를 좀 더 자세히 살펴보도록 하겠습니다.

표준 섀도우 맵 확장

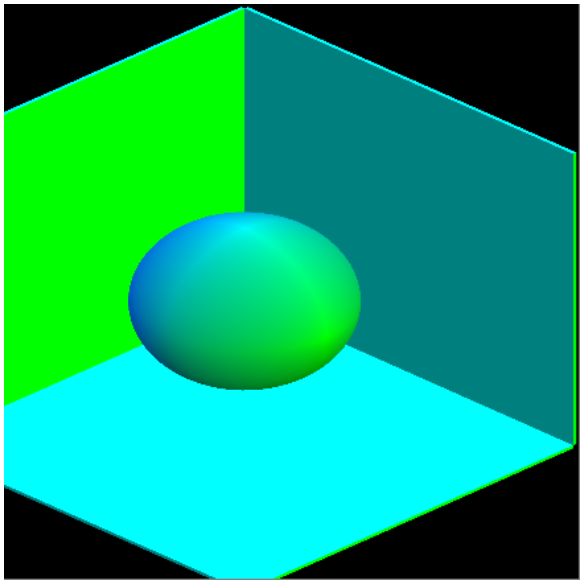
RSMs는 SSM을 확장하여 구현되기 때문에 당연하게도 SSM을 구현해 놓은 상태여야 합니다. 여기서는 SSM의 구현에 대해서는 다루지 않고 기존에 구현되어 있던 SSM을 RSMs에서 어떻게 확장하는지 알아보겠습니다. 이후 소개되는 RSMs 버퍼들은 다음과 같은 장면을 기준으로 생성되었습니다.



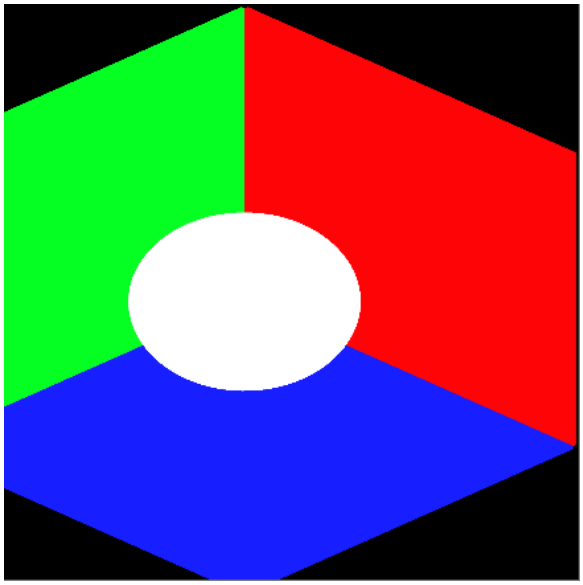
SSM은 조명의 시점에서 물체의 깊이를 저장했습니다. RSMs에서는 추가로 3가지 정보를 더 기록합니다.



첫 번째는 월드 좌표에서의 위치 (World Position)입니다. 메모리 절약을 위해서 깊이를 기록해서 월드 좌표를 재생성할 수도 있지만 구현의 편의성을 위해 RGBA_FLOAT 포맷의 텍스쳐에 월드 좌표를 직접 기록하였습니다.



두 번째는 월드 노멀 (World Normal)입니다. 흔히 보는 푸른 노멀 맵과는 사뭇 다른 모습인데 Octahedron normal vector encoding에 의해 R10G10B10A2 포맷의 텍스쳐에 기록됐기 때문입니다.



세 번째는 반사된 방사속 (Reflected Radiant Flux)입니다. 방사속은 단위 시간동안 전자파의 형태로 전달되는 에너지를 뜻합니다. 따라서 반사된 방사속은 조명에서 방출된 빛 에너지가 물체의 표면에 반사된 다음에 남은 빛의 에너지를 뜻합니다.

이제 실제 코드를 보면서 각 정보가 어떻게 기록하는지 살펴보겠습니다.

우선 버텍스 셰이더는 노멀을 픽셀 셰이더로 전달해야 하므로 위치만 사용하던 기존 코드에서 다음과 같이 노멀이 추가됩니다.

```
struct VS_INPUT
{
  float3 position : POSITION;
#if EnableRSMs == 1
  float3 normal : NORMAL;
#endif
  uint primitiveId : PRIMITIVEID;
};

struct VS_OUTPUT
{
  float4 position : POSITION;
#if EnableRSMs == 1
  float3 normal : NORMAL;
```

```
#endif
};

VS_OUTPUT main( VS_INPUT input )
{
  // ...
#if EnableRSMs == 1
  output.normal = mul( float4( input.normal, 0.f ), transpose( primitiveData.m_invWorldMatrix ) ).xyz;
#endif

  return output;
}
```

픽셀 셰이더에서는 MRT(Multi Render Target)를 사용하여 위의 3가지 정보를 텍스쳐에 기록하기 위해 셰이더 메인 함수의 반환 값을 구조체로 만들어 변수를 추가하고 필요한 정보를 기록합니다. 반사된 방사속 계산은 조명의 세기와 재질의 디퓨즈를 단순히 곱해서 계산할 수 있는데 이에 따라 추가적인 셰이더 리소스 바인딩이 필요합니다.

> 재질이 필요함에 따라서 섀도우 패스에서 하나의 메시가 여러가지 재질을 가진 경우에 대한 최적화를 적용할 수 없게 되어 결과적으로 섀도우 패스가 무거워질 수 있을 것 같습니다.

```
 struct PS_OUTPUT
{
  float depth : SV_TARGET0;
#if EnableRSMs == 1
  float3 worldPos : SV_TARGET1;
  float4 packedNormal : SV_TARGET2;
  float4 flux : SV_TARGET3;
#endif
};

PS_OUTPUT main( PS_INPUT input ) : SV_TARGET
{
  // ...
#if EnableRSMs == 1
  output.worldPos = input.worldPos;
  float3 enc = SignedOctEncode( normalize( input.normal ) );
  output.packedNormal = float4( 0.f, enc );
  output.flux = Diffuse * GetLight( LightIdx ).m_diffuse;
#endif

  return output;
}
```

간접광 계산

앞선 과정에서 섀도우 맵을 확장하여 3종류의 텍스쳐를 얻었습니다. RSMs에서 섀도우 맵의 각 텍셀은 점광원(Point light)으로 간주합니다.

> 이런 점광원을 Virtual Point Light(이하 VPL)라고 합니다.

광원의 크기가 매우 작을 때 w 방향으로 방출되는 복사 강도(Radiant intensity)는 다음과 같습니다.

$$I_p(w) = \Phi_p max\{0, \langle n_p | w \rangle\}$$

p : 점광원

Φ_p : 반사된 방사속

n_p : 노멀 벡터

$\langle|\rangle$: 내적(dot product)

따라서 노멀 벡터가 n인 표면의 점 x에 대한 점광원 p에 인한 방사 조도(Irradiance)는 다음과 같습니다.

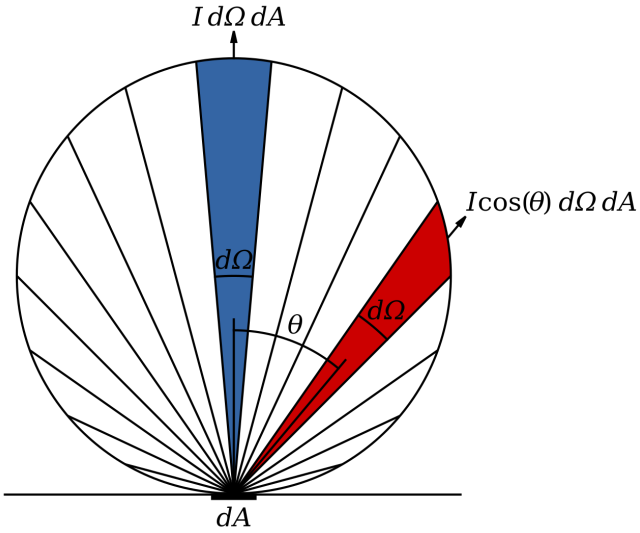
$$E_p(x,n) = \Phi_p \frac{max\{0, \langle n_p | x - x_p \rangle\} max\{0, \langle n | x_p - x \rangle\}}{||x - x_p||^4}$$

$||v||$: 벡터 v의 길이

위 식을 다음과 같이 분리해 보면 더 이해하기 쉬울 거라 생각됩니다.

$$E_p(x,n) = \Phi_p \frac{max\{0, \langle n_p | x - x_p \rangle\}}{||x - x_p||} \frac{max\{0, \langle n | x_p - x \rangle\}}{||x - x_p||} \frac{1}{||x - x_p||^2}$$

$\frac{max\{0, \langle n_p | x - x_p \rangle\}}{||x - x_p||}$는 점광원에서 x_p 위치로 얼마만큼의 빛이 반사되는지를 나타냅니다. 디퓨즈 표면은 램버트 코사인 법칙을(Lambert's cosine law)을 따르기 때문입니다.

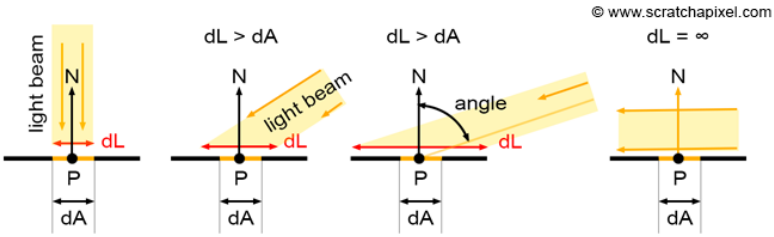


출처 : https://en.wikipedia.org/wiki/Lambert's_cosine_law

또한 $x - x_p$는 정규화된 벡터가 아니기 때문에 분모에 정규화를 위한 $||x - x_p||$가 포함됩니다.

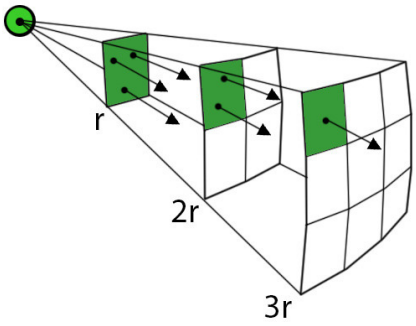
> 식을 천천히 살펴보면 결국 코사인을 구하는 것이라는 걸 알 수 있습니다.

$\frac{max\{0, \langle n | x_p - x \rangle\}}{||x - x_p||}$는 빛을 받는 표면에 도달하는 빛의 양을 나타냅니다. 빛의 받는 면적에 따라 밝기가 달라지기 때문에 이를 계산합니다.



출처 : https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-shading/diffuse-lambertian-shading.html

$\frac{1}{||x-x_p||^2}$ 는 거리에 따른 빛의 감쇄를 나타냅니다. 빛이 점광원에서 멀어질수록 동일 면적당 방사속의 비율은 떨어지며 이는 거리의 제곱에 반비례하게 됩니다.



출처 : http://brabl.com/light-attenuation/

> 구의 겉 넓이가 거리에 따라 증가하는 비율을 계산해 보면 됩니다.

이제 방사 조도 계산식을 이용하여 간접광을 계산하면 됩니다. 이 과정에서는 현재 위치를 기준으로 일정 범위를 섀도우 맵에서 샘플링하는데 Dachsbacher04에서 제시하는 샘플링 패턴은 다음과 같은 식으로 얻게 됩니다.

$$(s + r_{max}\xi_1 sin(2\pi\xi_2), t + r_{max}\xi_1 cos(2\pi\xi_2))$$

s, t : 섀도우 맵 텍스쳐 좌표

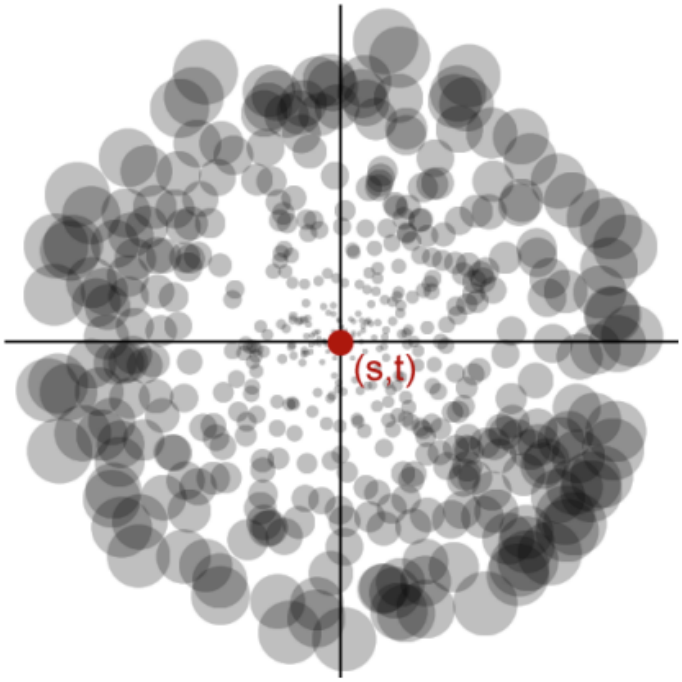
r_{max} : 샘플링 최대 범위(0 ~ 1)

ξ_1, ξ_2 : 균등 분포를 따르는 무작위 값

그리고 이렇게 생성된 샘플에 ξ_1^2 가중치를 주어 샘플링 밀도를 보상합니다.

> 거리가 멀어질수록 샘플링 밀도가 떨어지기 때문으로 생각됩니다.

이렇게 생성된 샘플링 패턴의 모습은 다음과 같습니다. 원의 크기가 가중치를 나타냅니다.



출처 : Dachsbacher04

지금까지 이야기한 내용을 실제 코드를 통해 정리하도록 하겠습니다.

우선 샘플링 패턴 생성 코드부터 보겠습니다.

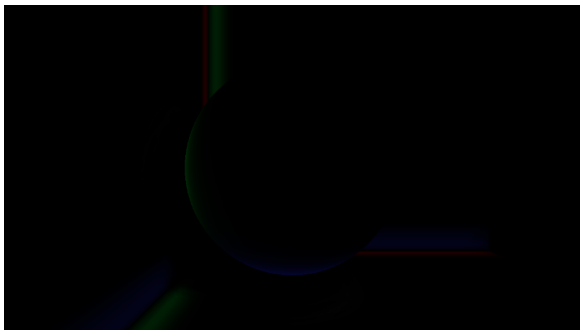
```
void RSMsRenderer::CreateSamplingPattern()
{
  // 최대 400개의 샘플링 패턴을 생성할 수 있습니다.
  constexpr uint32 MaxNumSamplingPattern = 400;
  uint32 numSamplingPattern = std::min( MaxNumSamplingPattern, DefaultRenderCore::RSMsNumSampling() );

  // 샘플링 패턴을 담을 버퍼를 생성합니다.
  agl::BufferTrait trait = {
    .m_stride = sizeof( Vector ),
    .m_count = numSamplingPattern,
    .m_access = agl::ResourceAccessFlag::GpuRead | agl::ResourceAccessFlag::GpuWrite,
    .m_bindType = agl::ResourceBindType::ShaderResource,
    .m_miscFlag = agl::ResourceMisc::None,
    .m_format = agl::ResourceFormat::R32G32B32_FLOAT
  };

  Vector samplingPattern[MaxNumSamplingPattern] = {};

  std::random_device rd;
  std::mt19937 mt( rd() );

  // 균등 분포로 0 ~ 1 사이의 숫자를 생성합니다.
  std::uniform_real_distribution<float> dis( 0, 1 );

  for ( uint32 i = 0; i < numSamplingPattern; ++i )
  {
    float xi1 = dis( mt );
    float xi2 = dis( mt );

    // Equation 3.
    samplingPattern[i][0] = xi1 * std::sin( 2.f * std::numbers::pi_v<float> * xi2 );
    samplingPattern[i][1] = xi1 * std::cos( 2.f * std::numbers::pi_v<float> * xi2 );
    samplingPattern[i][2] = xi1; // 가중치를 위한 xi1를 따로 저장해 놓습니다.
  }
```

```
  m_samplingPattern = agl::Buffer::Create( trait, samplingPattern );
  assert( m_samplingPattern != nullptr );

  m_samplingPattern->Init();

  // 샘플링 패턴의 개수와 최대 범위를 상수 버퍼를 통해 GPU에 전달합니다.
  RSMsConstantParameters params = {
    .m_numSamplingPattern = numSamplingPattern,
    .m_maxRadius = DefaultRenderCore::RSMsMaxSamplingRadius(),
  };
  m_constantParams.Update( params );
}
```

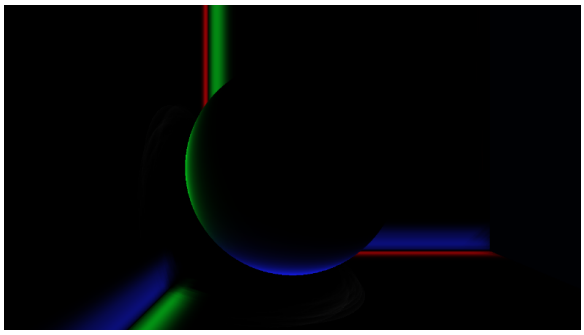
다음은 간접광 계산 픽셀 셰이더 코드입니다.

```
float4 main( PS_INPUT input ) : SV_TARGET
{
    // 현재 스크린 좌표 픽셀의 카메라 공간 위치를 계산합니다.
    float viewSpaceDistance = ViewSpaceDistance.Sample( BlackBorderSampler, input.uv ).x;
    float3 viewPosition = normalize( input.viewRay ) * viewSpaceDistance;

    // 카메라 공간 위치에서 월드 위치를 계산합니다.
    float4 worldPosition = mul( float4( viewPosition, 1 ), InvViewMatrix );
    worldPosition /= worldPosition.w;

    // 현재 스크린 좌표 픽셀의 섀도우 맵에서의 UV를 계산합니다.
    int cascadeIndex = SearchCascadeIndex( viewPosition.z );
    float4 shadowCoord = mul( worldPosition, ShadowViewProjection[cascadeIndex] );

    float2 shadowMapUV = shadowCoord.xy / shadowCoord.w;
    shadowMapUV.xy = shadowMapUV * float2( 0.5f, -0.5f ) + 0.5f;

    // 현재 스크린 좌표 픽셀의 월드 노멀을 얻어옵니다.
    float3 packedNormal = WorldNormal.Sample( BlackBorderSampler, input.uv ).yzw;
    float3 worldNormal = SignedOctDecode( packedNormal );

    // 캐스케이드 단계에 따라서 샘플링 거리를 조절합니다.
    float cascadeScale = CascadeConstants[0].m_zFar / CascadeConstants[cascadeIndex].m_zFar;
    float3 indirectLight = 0.f;
    [loop]
    for ( uint i = 0; i < NumSamplingPattern; ++i )
    {
        // 섀도우 맵을 샘플링할 UV좌표를 계산합니다.
        float3 samplingOffset = SamplingPattern[i];
        float3 uv = float3( shadowMapUV + MaxRadius * cascadeScale * samplingOffset.xy, cascadeIndex );

        // 샘플링 위치의 월드 워치, 노멀, 반사된 방사속을 얻어옵니다.
        float3 positionP = RSMsWorldPosition.SampleLevel( BlackBorderSampler, uv, 0 ).xyz;
        float3 normalP = SignedOctDecode( RSMsNormal.SampleLevel( BlackBorderSampler, uv, 0 ).yzw );
        float3 fluxP = RSMsFlux.SampleLevel( BlackBorderSampler, uv, 0 );

        // Equation 1.
        // 방사 조도 계산식
        float3 irrandiance = fluxP * ( ( max( 0.f, dot( normalP, worldPosition - positionP ) )
                  * max( 0.f, dot( worldNormal, positionP - worldPosition ) ) )
                  / pow( length( worldPosition - positionP ), 4 ) );

        // 샘플링 거리에 따른 가중치 반영
        irrandiance *= samplingOffset.z * samplingOffset.z;
        indirectLight += irrandiance;
    }

    return float4( indirectLight, 1.f );
}
```

이렇게 다음과 같은 간접광을 얻을 수 있습니다.



실제 결과



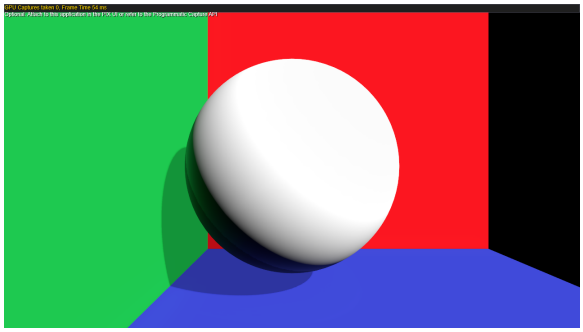
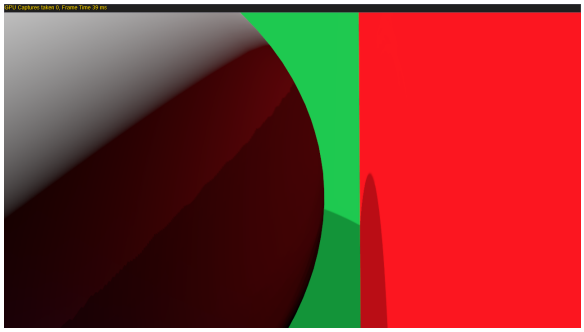
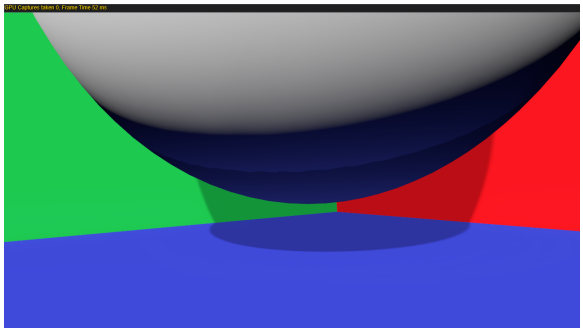
밝기를 증가시킨 결과

최종 합성

이제 간접광 계산 결과를 최종 장면에 합성하면 됩니다. 제 구현에서는 포워드 렌더링으로 직접광을 계산하는 과정에서 간접광을 합성하도록 하였습니다.

```
float4 main( PS_INPUT input ) : SV_Target0
{
  GeometryProperty geometry = (GeometryProperty)0;
  geometry.worldPos = input.worldPos;
  geometry.viewPos = input.viewPos;
  geometry.normal = input.normal;
  // 전체 화면 UV를 계산
  geometry.screenUV = ( input.projectionPos.xy / input.projectionPos.w ) * float2( 0.5f, -0.5f ) + 0.5f;

  return CalcLight( geometry );
}

float4 CalcLight( GeometryProperty geometry )
{
  // 직접광 계산 부분 생략

  // 간접광 적용
#if EnableRSMs == 1
  lightColor += MoveLinearSpace( Diffuse ) * IndirectIllumination.Sample( LinearSampler, geometry.screenUV );
#endif

  return saturate( lightColor );
}
```

최종 결과 장면은 다음과 같습니다.


느낀 점

RSMs는 비교적 간단한 구현으로 그럴싸한 간접광을 표현할 수 있었습니다. 하지만 간접광 계산과정에서 유효하지 않은 샘플을 샘플링하는 경우에 불필요한 계산이 이뤄지고 이에 따라 결과물이 안정적이지 않은 경우가 보였습니다. 안정적인 결과물을 얻으려면 샘플링 횟수가 늘어나야 하는데 루프가 길어질수록 성능 저하가 눈에 띄게 관찰되어 높은 반응성을 위한 애플리케이션에 사용하기에는 아쉬운 감이 있습니다. 이를 해결하기 위해 저 해상도에서 간접광을 계산하고 업 샘플링하는 방식을 논문에서 소개하고 있습니다.

다만 저는 해당 방식을 구현하지 않았는데요. RSMs는 여기서 소개한 것에 그치지 않고 Light Propagation Volume(이하 LPV)이라는 또 다른 동적 GI의 기반이 되는데 제가 최종적으로 구현하고 싶은 대상이 LPV이기 때문에 이 정도에서 정리하였습니다. 그래서 다음 기술 공유는 이변이 없다면 LPV가 될 것으로 생각됩니다.

마치며

준비한 내용은 여기까지입니다.
전체 코드는 아래의 링크를 참고하시면 됩니다.

Commits · xtozero/SSR

Screen Space Reflection. Contribute to xtozero/SSR development by creating an account on GitHub.

https://github.com/xtozero/SSR/commits/reflective_shadow_maps

xtozero/**SSR**

Screen Space Reflection

1 Contributor 0 Issues 13 Stars 1 Fork

Reference

http://www.klayge.org/material/3_12/GI/rsm.pdf : Dachsbacher04