Screen Space Reflection

Screen Space Reflection?

Screen Space Reflection

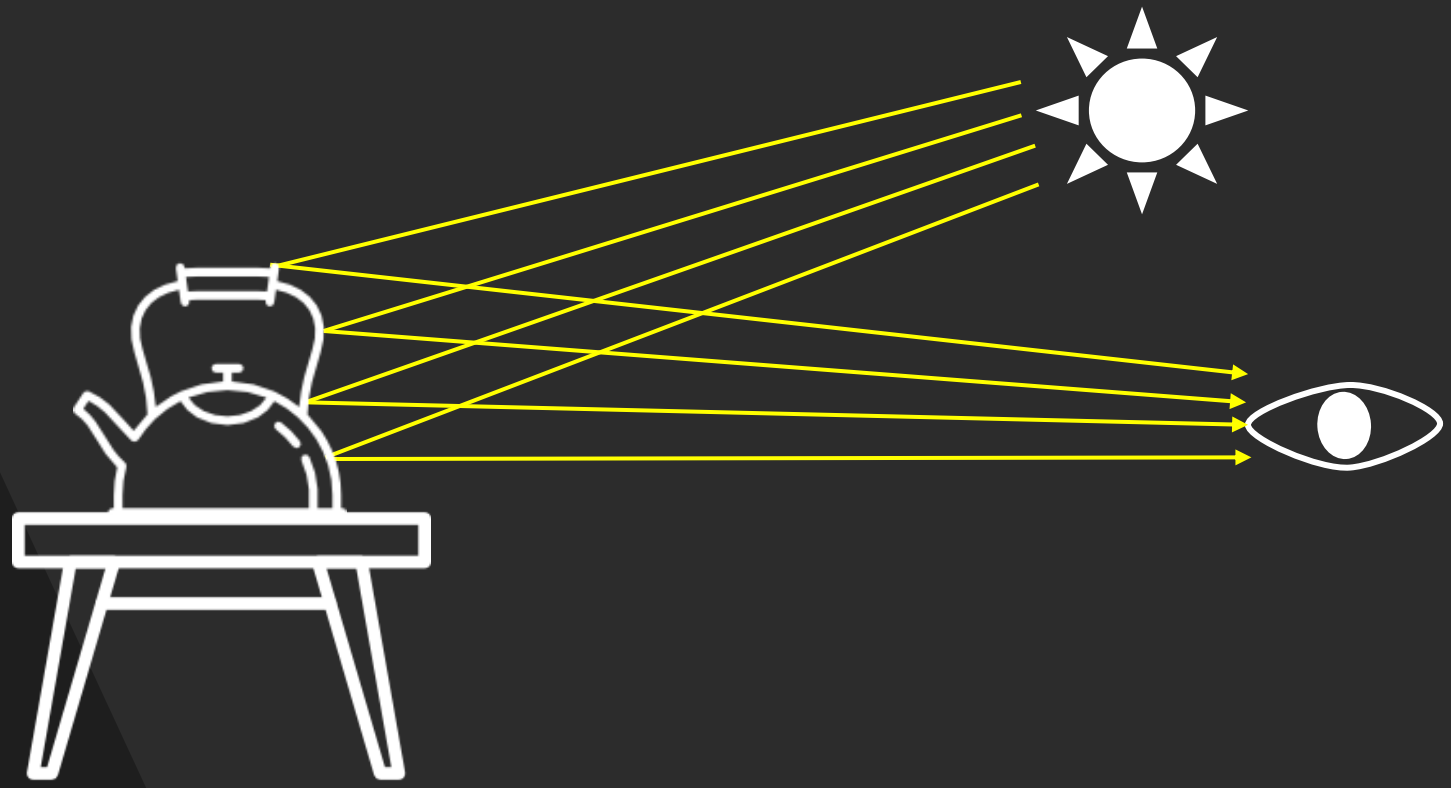
: 화면

: 공간

: 반사

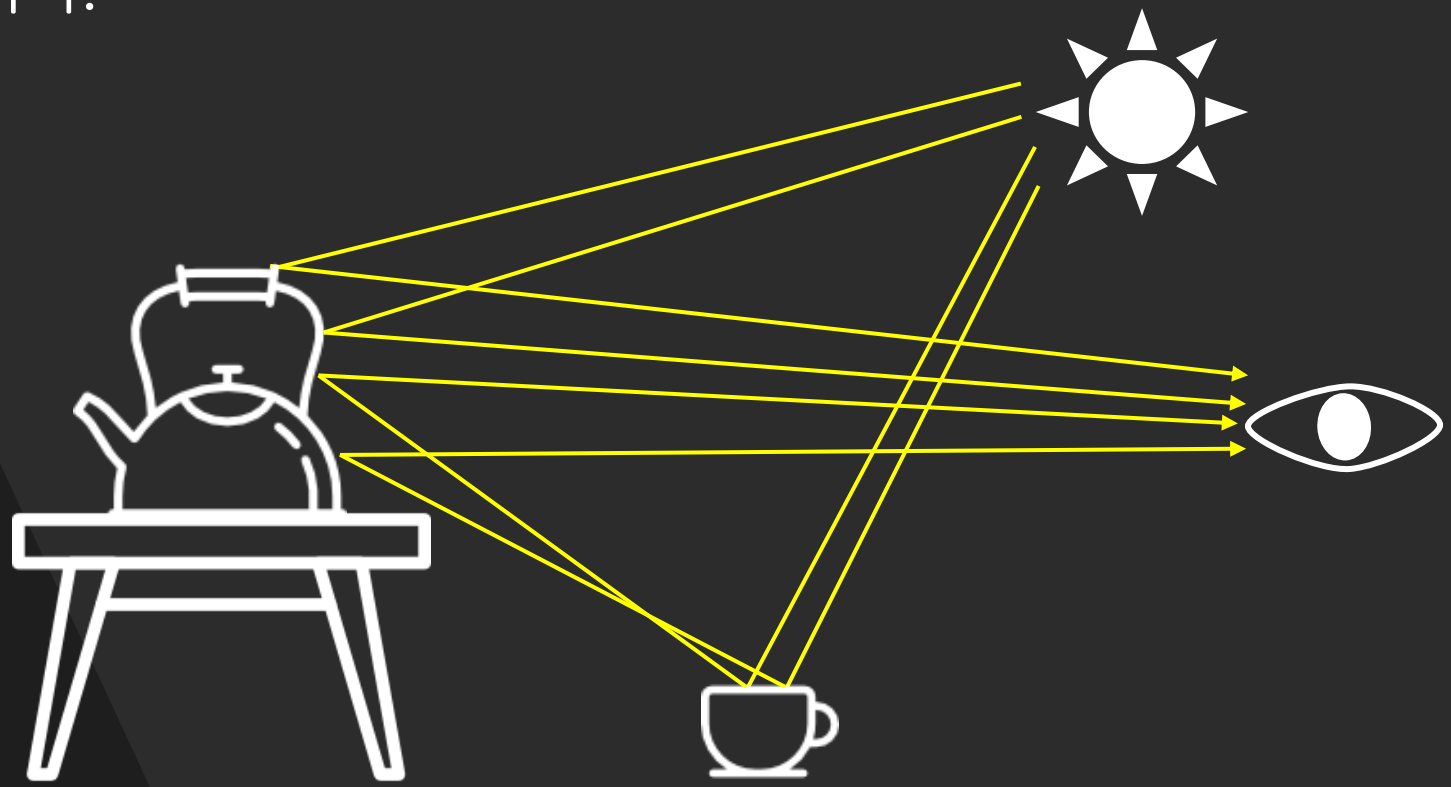
Reflection?

사람은 반사된 빛을 통해서 사물을 봅니다.



Reflection?

그런데 눈으로 반사된 빛은 광원에서 나온 빛 뿐만이 아니라 다른 물체에서 반사된 빛일 수 있습니다.

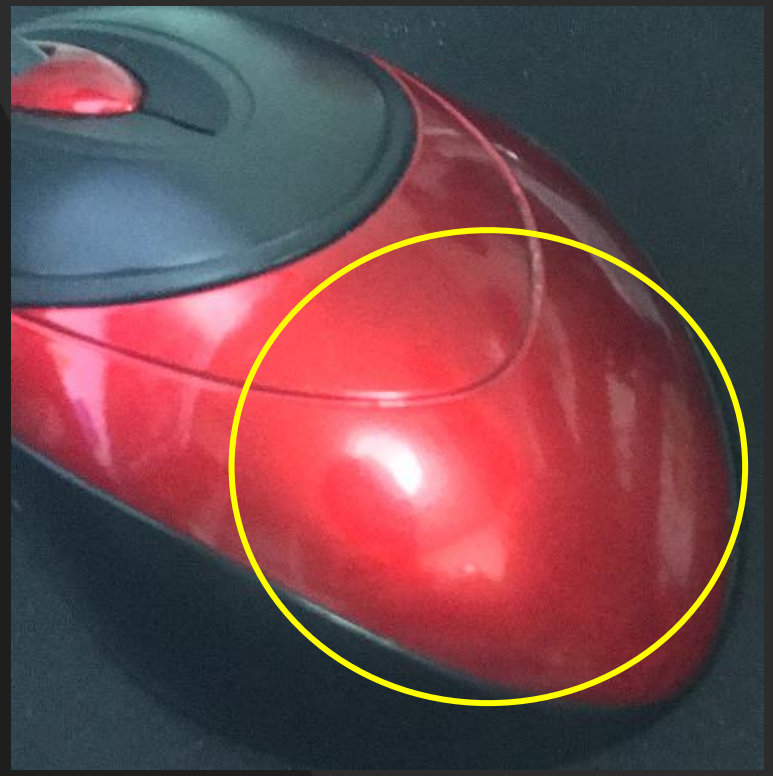


Reflection?

거울에서 이런 현상을 가장 잘 관찰할 수 있습니다.

Reflection?

거울 뿐만 아니라 다양한 사물에서도 관찰할 수 있습니다.



Reflection?

3D 애플리케이션에서 이런 반사를 어떻게 표현할 수 있을까요?

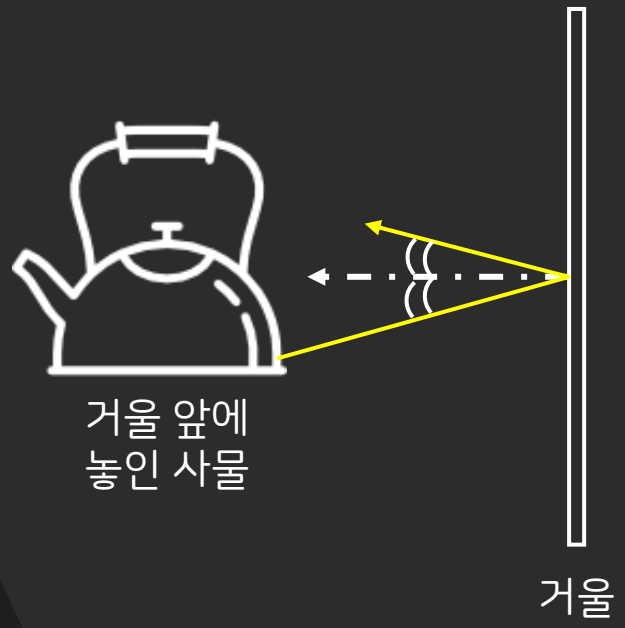
다음과 같은 임의의 평면에 놓인 하나의 거울을 생각해 봅시다.

거울 앞에
놓인 사물

거울

Reflection?

거울과 같은 매끄러운 표면에 들어온 빛은 표면의 법선 방향에 대해서 입사각과 동일한 각도로 반사됩니다.



Reflection?

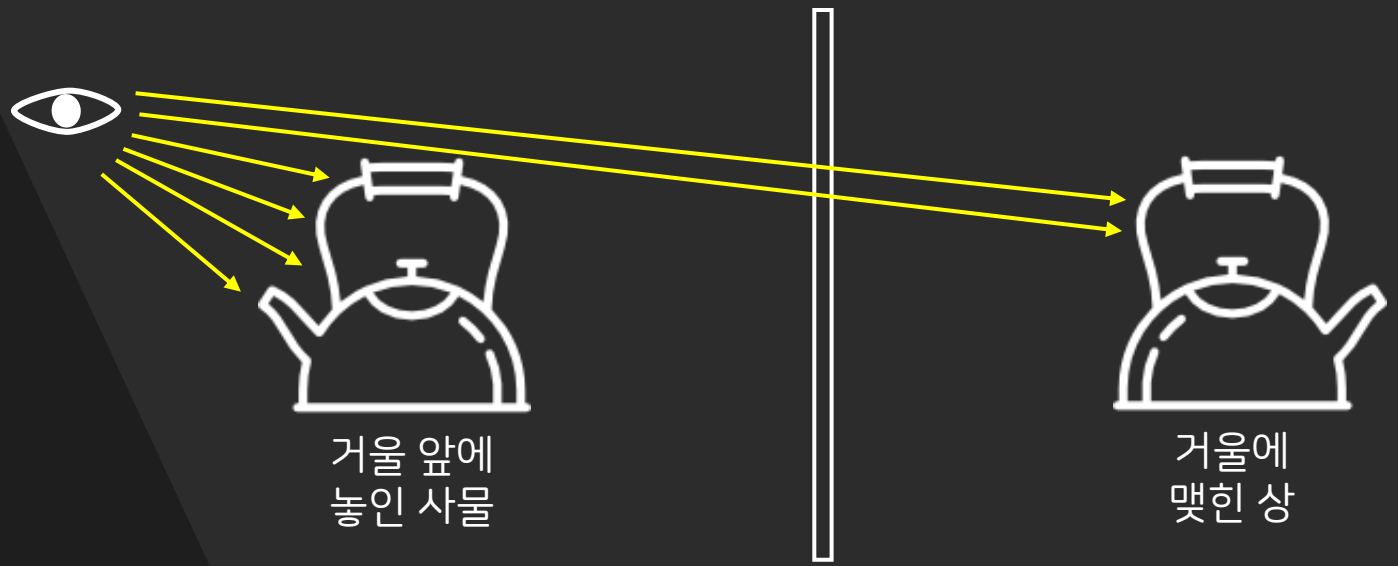
그 결과 거울 평면을 대칭으로 사물의 상이 맺히게 됩니다.

3D 애플리케이션에서 장면에 거울이 하나만 있다면 거울에 맺히는 상의 위치로 사물을 이동하여 한번 더 그리면 됩니다.

Reflection?

즉 거울 이동 행렬을 장면의 모든 물체에 적용하여 한번씩 더 렌더링하면 됩니다.

다만 거울은 앞뒤가 반전되므로 카메라를 기준으로 뒷면인 면이 거울의 상으로 그려질 수 있습니다. 따라서 winding order를 고려해야 합니다.



Reflection?

이제 3D 애플리케이션에서 거울에 반사된 상을 표현할 수 있습니다.

그럼 한 장면에 거울이 여러 개라면 어떻게 하면 될까요?
=> 거울의 개수 만큼 반사 행렬을 만들어 물체를 여러 번 그리면 됩니다.

즉 반사 행렬을 사용한 방식은 장면의 물체 개수 X 거울의 개수 만큼 더 렌더링을 더 해야 합니다.

거울의 개수에 따라 Draw call이 증가하게 됩니다.

Reflection?

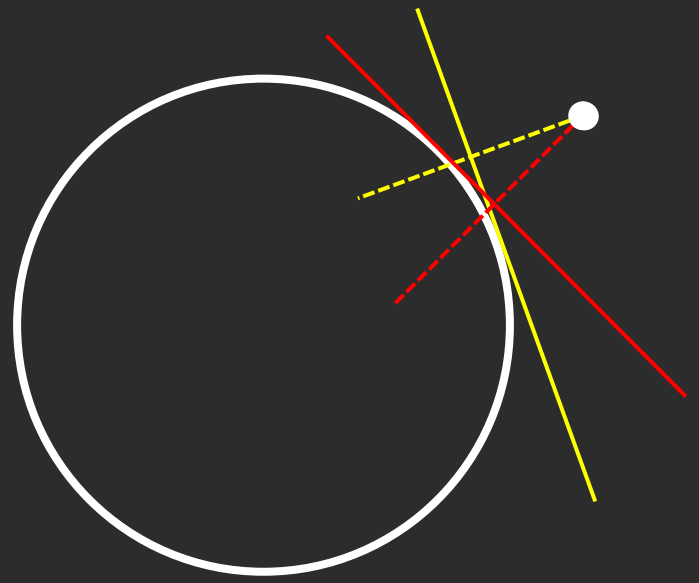
그리고 이런 물체는 표현할 수 있을 까요?

출처 : https://en.wikipedia.org/wiki/Specular_reflection

Reflection?

반사 행렬을 사용한 방식은 상이 맺히는 위치에 물체를 여러 번 그립니다.

구와 같이 여러 평면이 존재하는 기하 구조는 물체를 여러 번 겹쳐 그려야 합니다.



Environment Mapping

이 경우 환경 매핑을 사용할 수 있습니다.

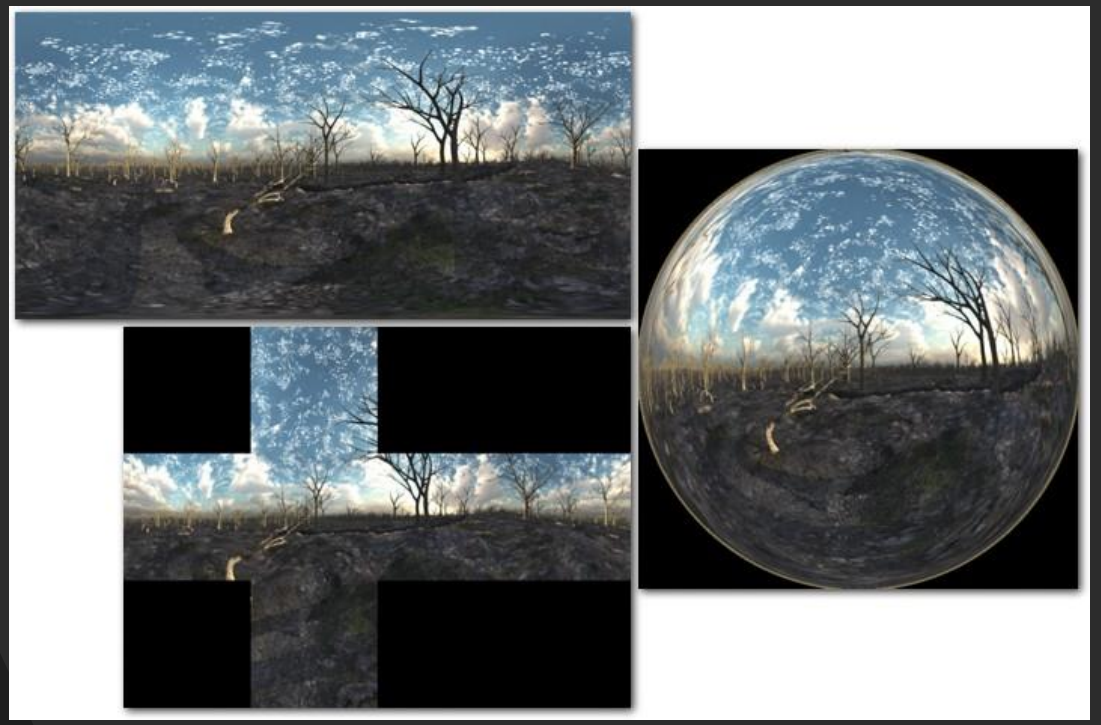
물체를 기준으로 주변의 환경을 렌더링하고 이 정보를 바탕으로 반사된 상을 계산하는 것입니다.

구글 스트리트 뷰와 같은 파노라마 사진을 생각하면 됩니다.

출처 : https://www.google.co.kr/intl/ko/streetview/apps/

Environment Mapping

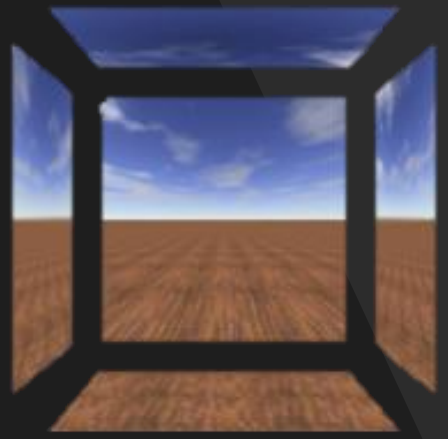
주변 환경은 다양한 형식의 텍스처로 사전에 저장할 수 있고 latitude/longitude, mirrored ball, horizontal cross 등의 형식으로 저장될 수 있습니다.

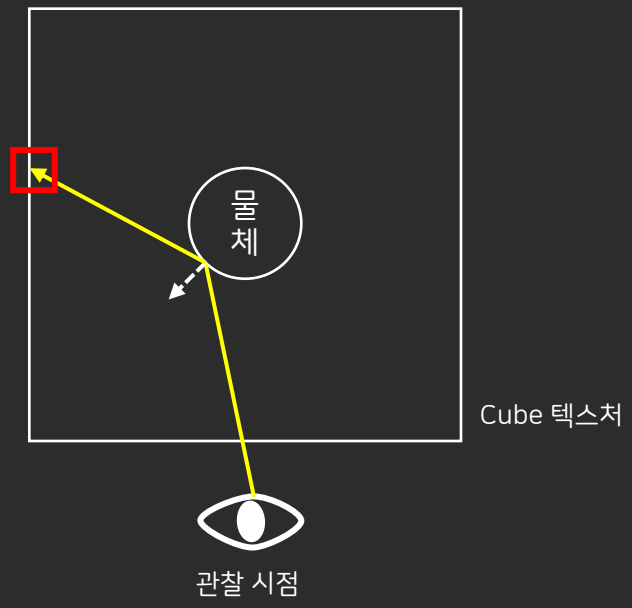


출처 : http://spiralgraphics.biz/genetica/help/index.htm?environment_maps_explained.htm

Environment Mapping

그 중에서도 정육면체 형태의 Cube 텍스처를 주로 사용하는데 간단한 셰이더 코드로 손쉽게 환경 매핑을 구현할 수 있습니다.



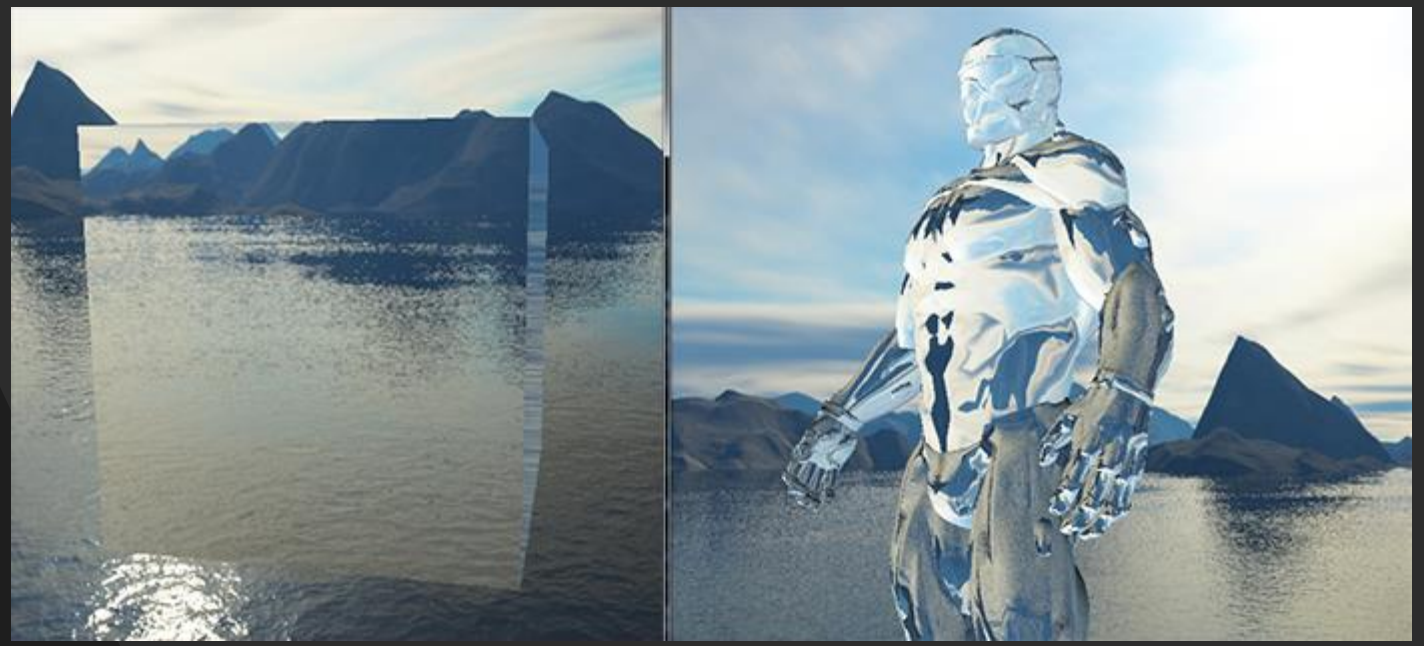



```
TextureCube cubeTex : register( t0 );

float4 main( PS_INPUT input ) : SV_Target
{
    return cubeTex.Sample( baseSampler, 반사 벡터);
}
```

Environment Mapping

반사 벡터를 통해 Cube 텍스처를 샘플링하면 다음과 같이 반사도가 높은 표면을 3D 애플리케이션에서 표현할 수 있게 됩니다.



출처 : https://learnopengl.com/Advanced-OpenGL/Cubemaps

Environment Mapping

환경 매핑은 물체에 거울과 같은 높은 반사도를 가지는 다양한 표면이 있을 경우에 유용하게 사용될 수 있습니다.

하지만 애플리케이션내에서 실시간으로 움직이는 물체를 표현하기 위해서는 반사 행렬을 사용 했던 것처럼 주변 환경을 매번 다시 그려줘야 합니다.

Cube 텍스처를 사용할 경우 6면의 텍스처 X 물체의 개수 만큼의 Draw Call이 추가 될 수 있습니다.

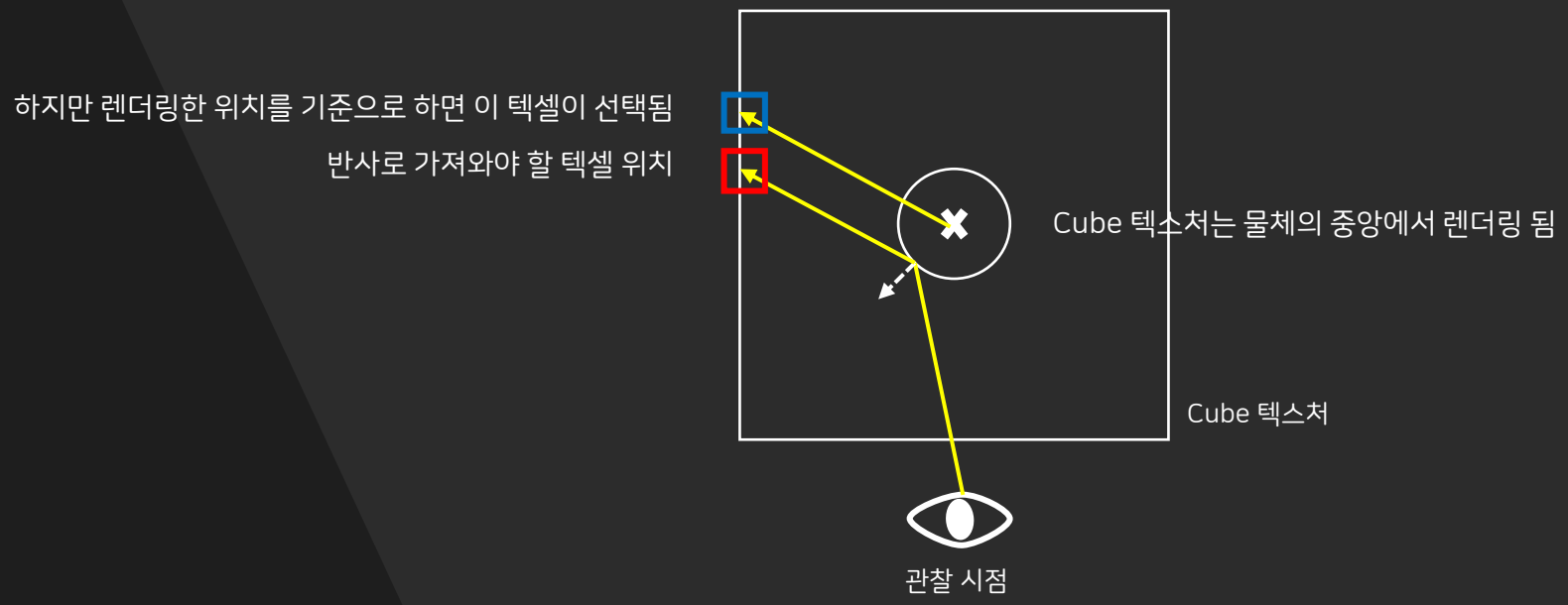
Environment Mapping

또 다음과 같은 문제가 발생할 수 있습니다.

출처 : http://www.cse.chalmers.se/edu/year/2017/course/TDA361/Advanced%20Computer%20Graphics/Screen-space%20reflections.pdf

Environment Mapping

이 문제는 주변 환경을 렌더링한 위치가 실제 반사가 일어나는 위치와 다르기 때문에 발생하는 현상입니다.



Environment Mapping

즉 환경 매핑은 물체에 반사되는 주변환경이 물체에서 무한한 거리만큼 떨어져 있을 경우에 적절한 효과를 제공합니다.

마지막으로 환경 매핑은 자기 자신에 대한 반사를 표현하지 못합니다.

주전자의 주둥이 부분이 보이지 않음

출처 : http://schorsch.efi.fh-nuernberg.de/roettger/index.php/Computergrafik/EnvironmentMapping

Screen Space Reflection

반사 행렬을 사용하는 방법은 반사 행렬이 특정 표면에 제한되므로 다른 평면에 반사되는 물체를 그리기 위해서는 행렬을 다시 구해서 물체를 중복으로 그려야 합니다.

환경 매핑은 실시간 반사를 위해서 여전히 물체를 다시 그려야 하고 반사되는 물체가 반사면에 근접해 있을 때는 괴리감이 발생할 수 있습니다.

이런 기존 기법들의 단점을 어느정도 보완한 기법이 바로 Screen Space Reflection 입니다.

Screen Space Reflection

Screen Space Reflection의 핵심을 파악하기 위해서 다음 사진을 살펴보겠습니다.
수면에 건물이 깔끔하게 반사되어 보이는 사진입니다.

Screen Space Reflection

이 사진을 잘 살펴보면 수면에 반사하여 보이는 건물들을 사진에서 모두 찾을 수 있다는 걸 알 수 있습니다.

Screen Space Reflection

즉 화면 공간(Screen Space)의 데이터를 통해서 반사를 계산할 수 있습니다.

이번 프레임에 그린 장면의 데이터를 재사용하면 반사를 위해서 물체를 여러 번 그리지 않아도 되므로 장면의 복잡도에 영향을 받지 않습니다.

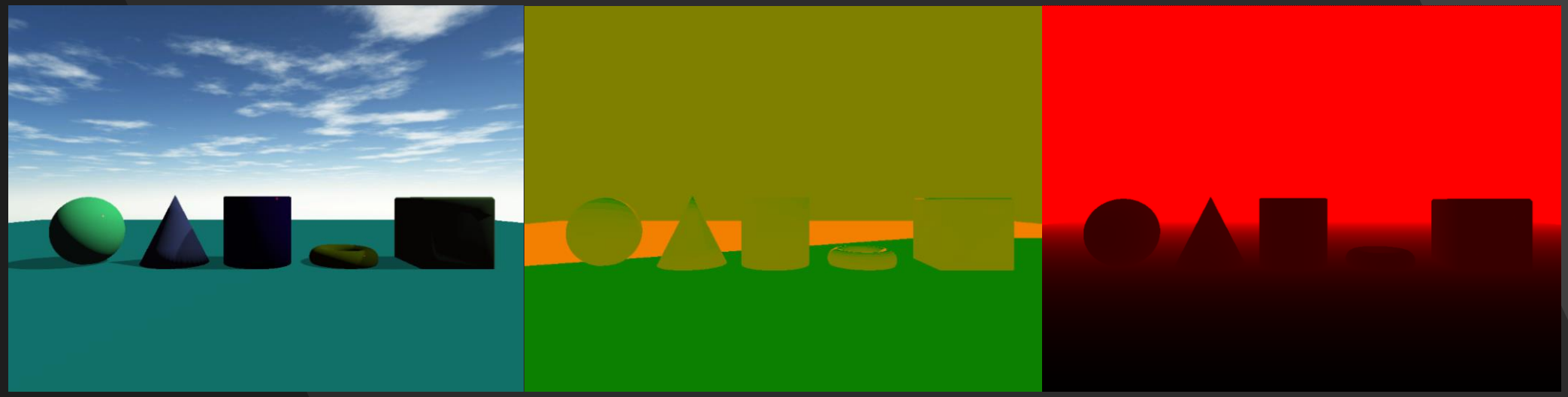
또한 장면은 매프레임 갱신되야 하기 때문에 실시간 반사를 위한 추가 작업이 필요하지 않습니다.

자기 자신에 대한 반사도 가능하며 특정 표면에 제한되지 않습니다.

Screen Space Reflection

화면 공간의 데이터를 통해서 반사를 계산하려면 3가지 정보를 알고 있어야 합니다.

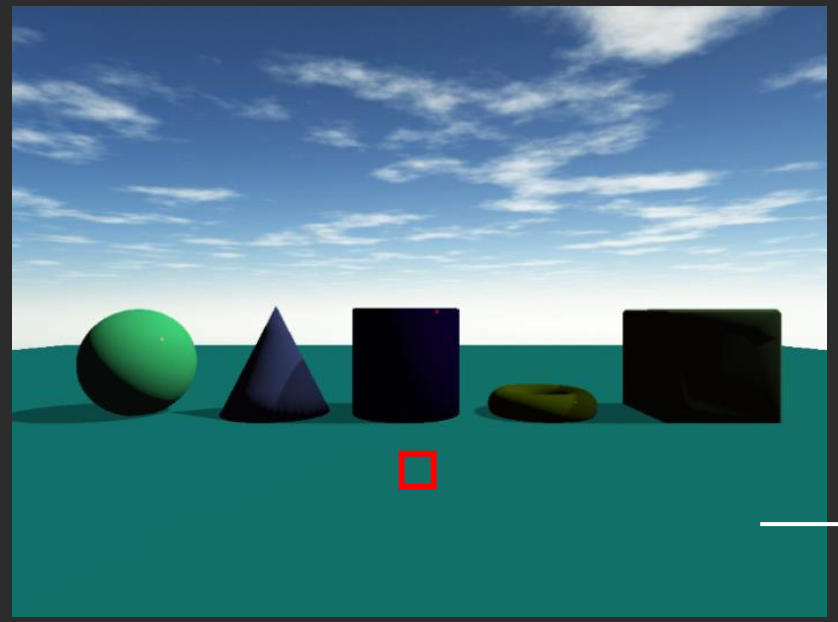
1. 장면의 색상 정보
2. 장면의 법선 정보
3. 장면의 위치 정보



Screen Space Reflection

Screen Space Reflection으로 아래와 같은 픽셀 위치의 반사를 계산해보겠습니다.

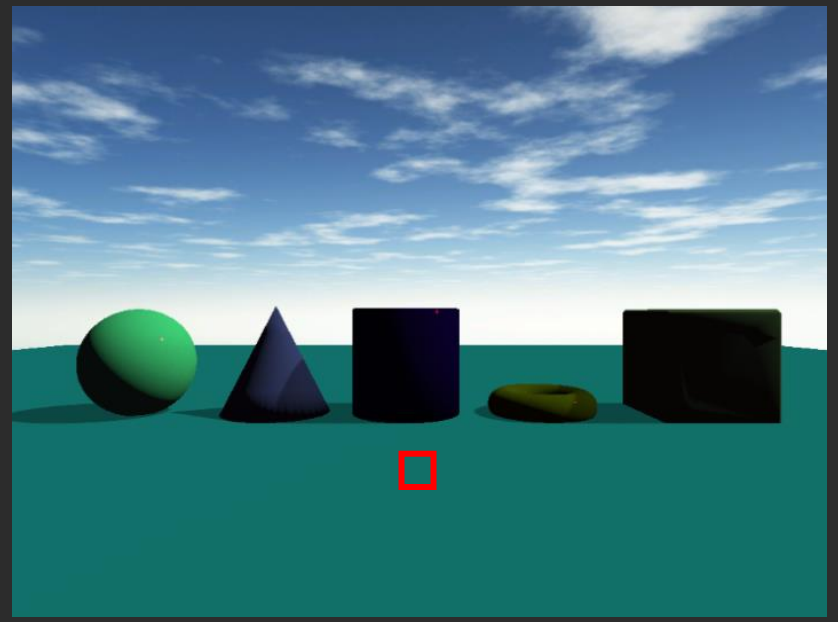
카메라에서 해당 픽셀 까지의 광선이 표면에 반사되어 어느 물체에 부딪히는지를 추적하면 됩니다.



Screen Space Reflection

카메라 공간에서 카메라의 위치는 (0, 0, 0) 이므로 해당 픽셀로 광선이 진행하는 방향은 픽셀의 카메라 공간 위치가 됩니다.

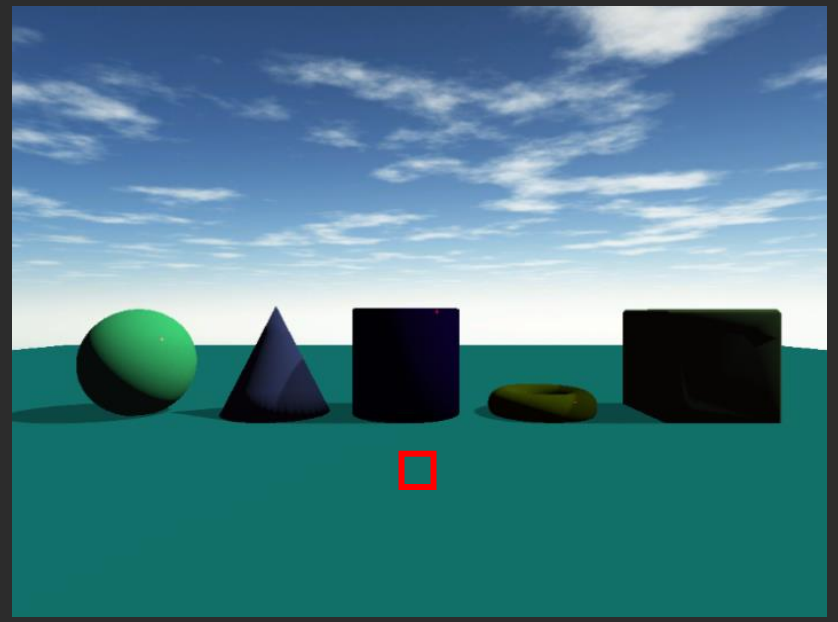
이를 장면의 위치 정보를 이용하여 얻어 냅니다.

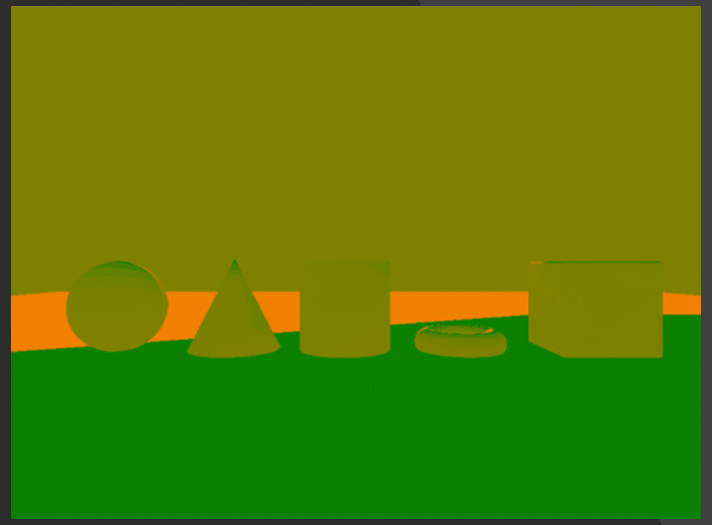


Screen Space Reflection

이제 해당 픽셀의 법선 벡터를 알면 반사 벡터를 구할 수 있습니다.

```
float3 reflectVec = reflect( incidentVec, viewNormal );
```

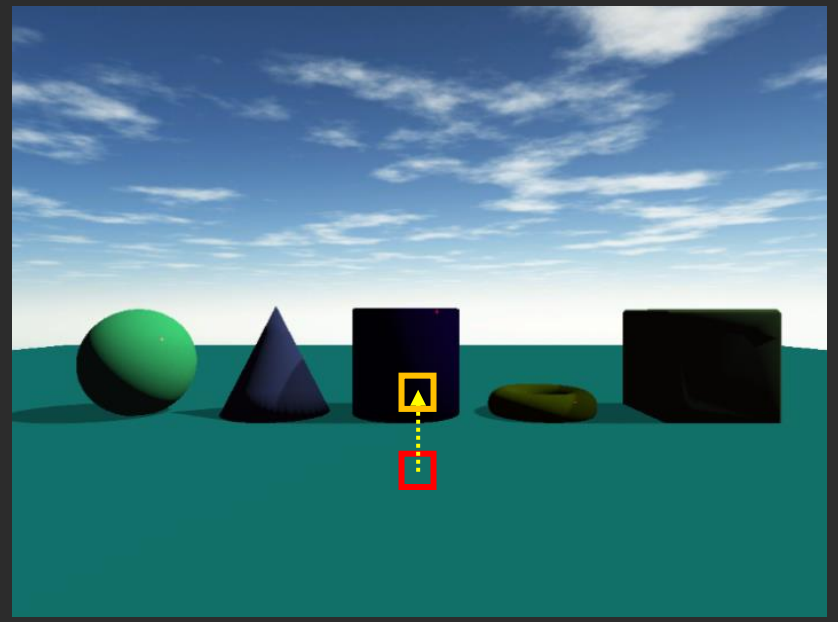


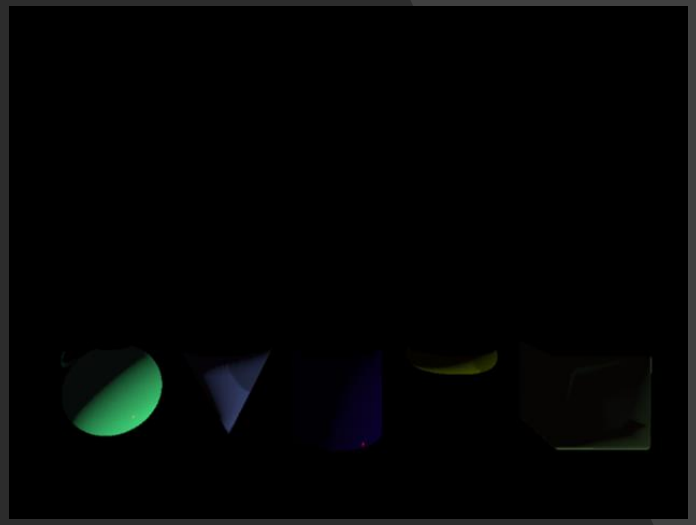


Screen Space Reflection

이제 반사 벡터의 방향으로 Ray Marching을 통해서 광선이 충돌하는 위치를 알아내고 해당 위치의 텍셀을 가져오면 됩니다.

충돌하는 위치는 어떻게 알아 낼 수 있을까요?





Ray Marching?

광선과 어떤 표면의 충돌점을 비교적 쉽게 구할 수 있는 방법을 한번 살펴보겠습니다.

어떤 광선이 구에 충돌한다면 그 충돌점은 광선과 구를 함수로 표현한 다음 광선의 함수를 구의 함수에 대입하는 것으로 계산할 수 있습니다.

광선을 함수로 표현 : $R(t) = P + td\ (t \geq 0)$

구를 함수로 표현 : $(X - C) \cdot (X - C) = r^2$

$(P + td\ - \mathrm{C}) \cdot (P + td\ - \mathrm{C}) = r^2$

$(m + td) \cdot (m + td) = r^2$ ($P - C$를 m으로 치환)

$(d \cdot d)t^2 + 2(m \cdot d)t + (m \cdot m) = r^2$ (전개)

$\boldsymbol{t^2 + 2(m \cdot d)t + (m \cdot m) - r^2 = 0\ (d \cdot d = 1)}$

근의 공식을 통해서 $\boldsymbol{t}$ 를 구할 수 있고 이를 광선의 함수에 대입하면 위치

Ray Marching?

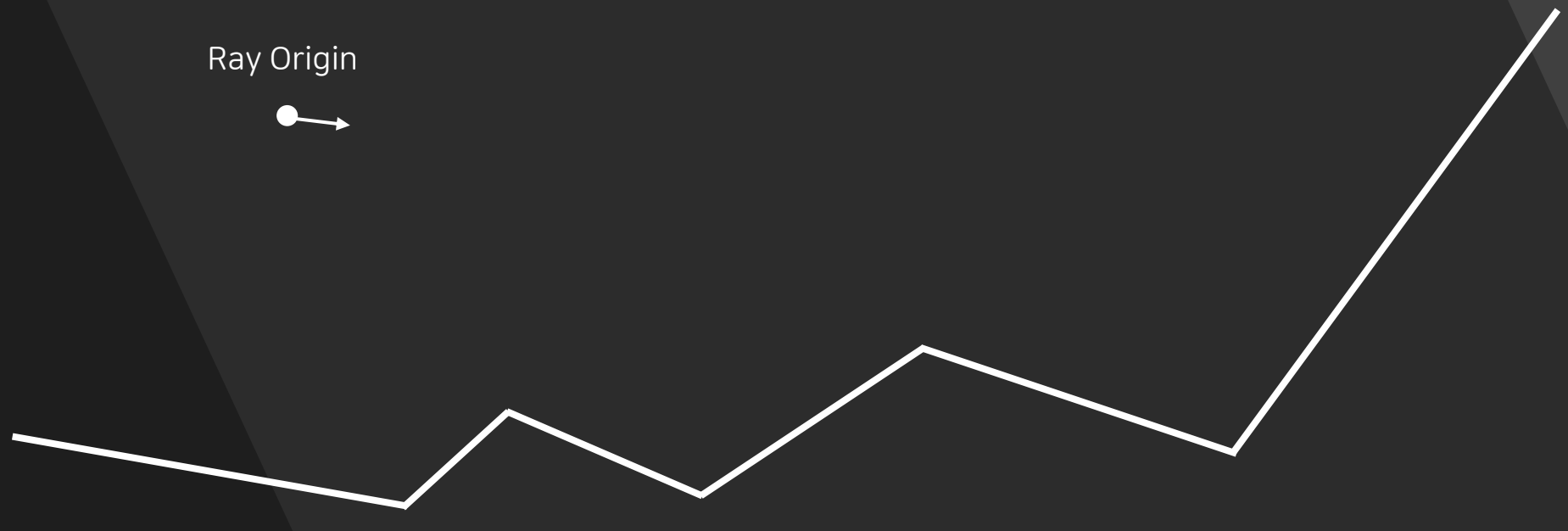
이렇게 특정 표면을 함수로 표현할 수 있는 경우에는 충돌점을 비교적 쉽게 찾아 낼 수 있습니다.

하지만 이미 그려진 복잡한 장면의 표면은 함수로 표현하기 어렵습니다.

Ray Marching으로 표면 함수가 쉽게 풀리지 않을 경우에 표면과 광선의 충돌점을 찾아낼 수 있습니다.

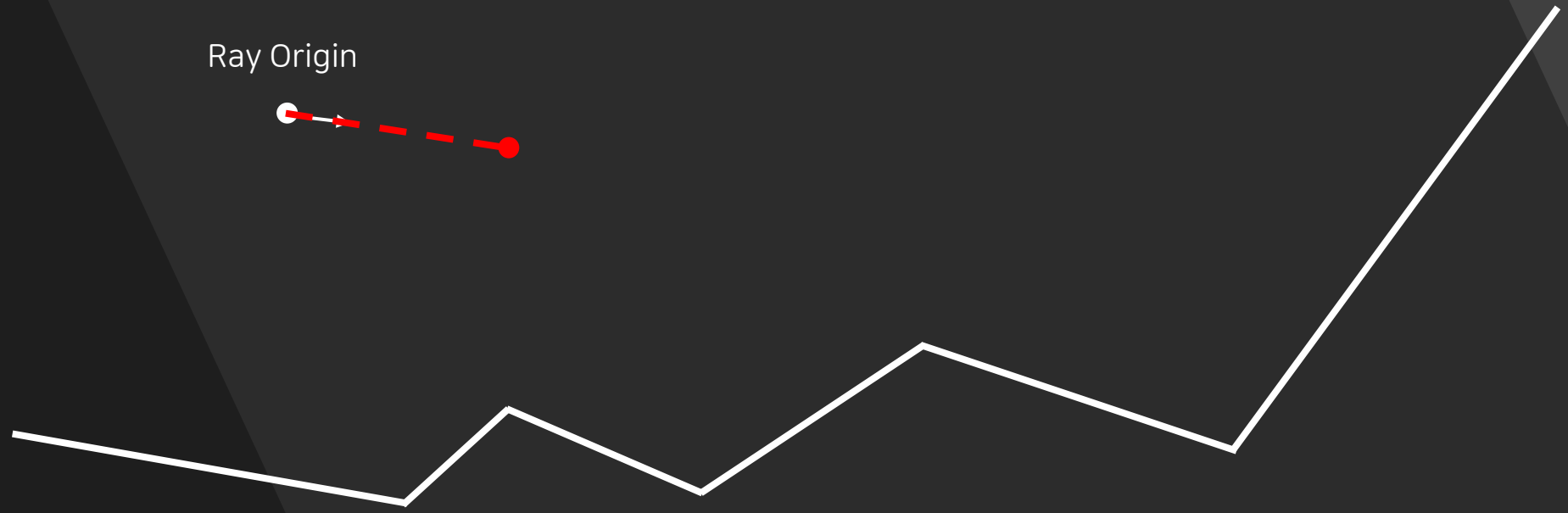
Ray Marching?

다음과 같은 표면에서 Ray Marching을 사용하여 광선의 충돌점을 찾아보겠습니다.



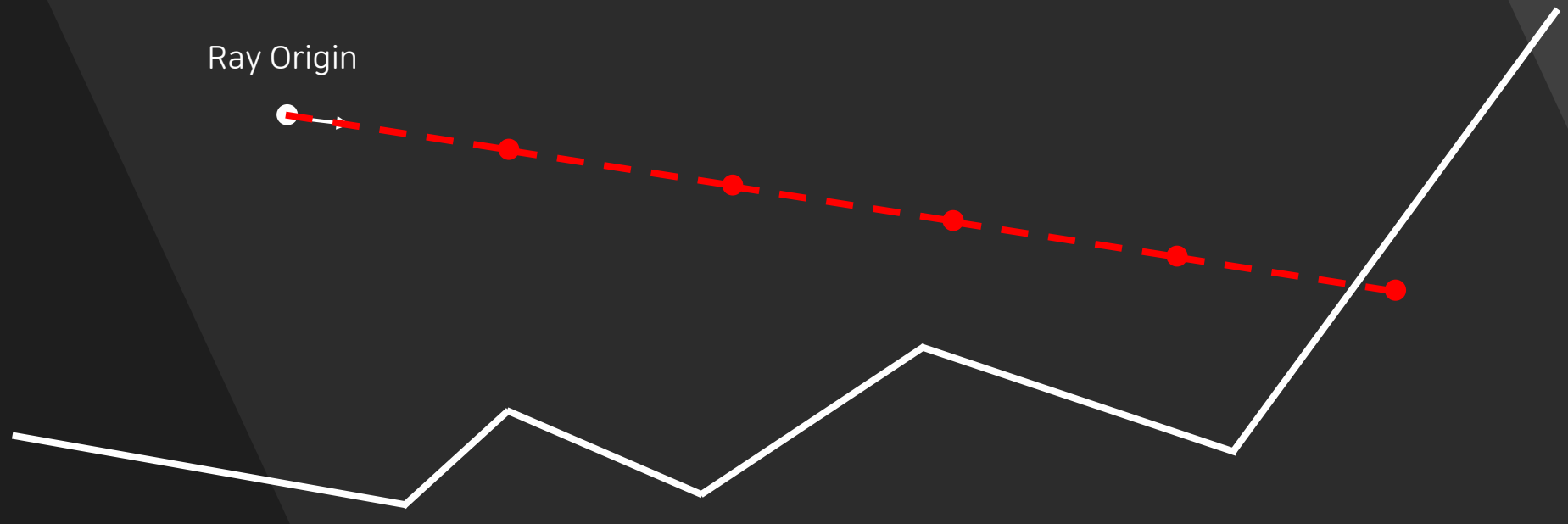
Ray Marching?

광선을 진행 방향으로 일정거리 만큼 진행합니다.
해당 위치가 표면과 충돌했는지 검사합니다. 이 단계에서는 충돌하지 않았습니다.



Ray Marching?

표면과 충돌 할 때 까지 반복합니다.
이제 표면과 충돌하였습니다.



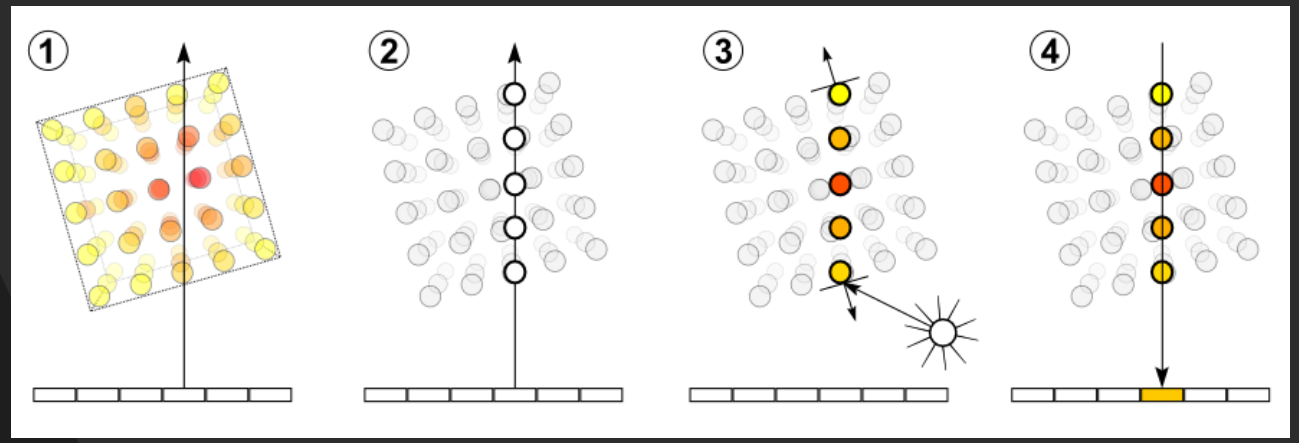
Ray Marching?

알아낸 충돌위치에서 추가적으로 이진탐색을 수행하여 정확도를 높일 수 있습니다.

Ray Marching?

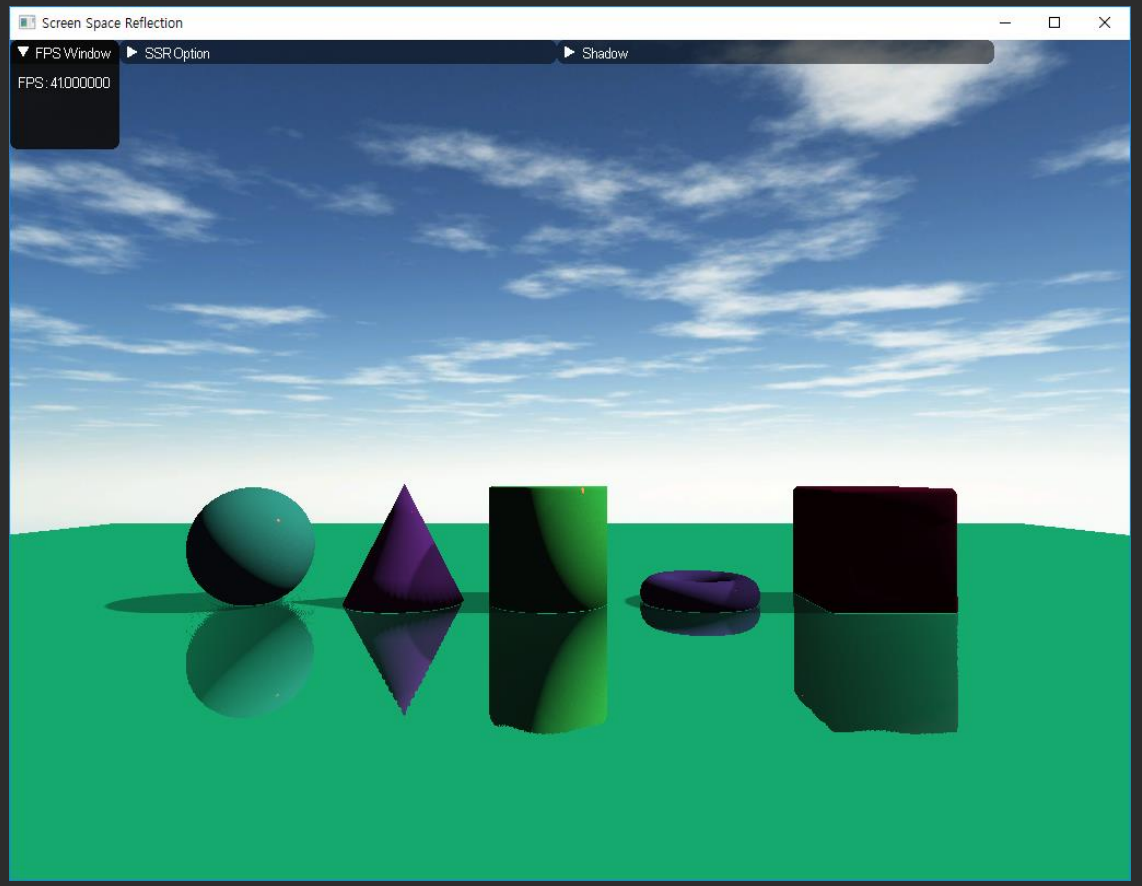
이런 식으로 행진하듯이 광선의 진행 방향으로 일정 거리만큼 이동시켜 가며 충돌점을 찾아내는 방식을 Ray Marching이라고 합니다.

충돌점을 찾아내는 것 외에도 Ray Marching은 균일하지 않은 볼륨을 가진 투명한 물체를 그릴 때도 사용됩니다.



출처 : https://en.wikipedia.org/wiki/Volume_ray_casting

Screen Space Reflection Using Ray Marching



Screen Space Reflection Using Ray Marching

```
float4 main( PS_INPUT input ) : SV_TARGET
{
    float3 incidentVec = normalize( input.viewPos );
    float3 viewNormal = normalize( input.viewNormal );

    float3 reflectVec = reflect( incidentVec, viewNormal );
    reflectVec = normalize( reflectVec );
    reflectVec *= g_rayStepScale;

    float3 reflectPos = input.viewPos;

    float thickness = g_maxThickness / g_FarPlaneDist;

    [loop]
    for ( int i = 0; i < g_maxRayStep; ++i )
    {
        float3 texCoord = GetTexCoordXYLinearDepthZ( reflectPos );
        float srcdepth = depthbufferTex.SampleLevel( baseSampler, texCoord.xy, 0 ).x;

        float depthDiff = texCoord.z - srcdepth;
        if ( depthDiff > g_depthbias && depthDiff < thickness )
        {
            float4 reflectColor = BinarySearch( reflectVec, reflectPos );

            float edgeFade = 1.f - pow( length( texCoord.xy - 0.5f ) * 2.f, 2.f );
            reflectColor.a *= pow( 0.75f, (length( reflectPos - input.viewPos ) / g_maxRayLength) ) * edgeFade;
            return reflectColor;
        }
        else
        {
            reflectPos += ( i + Noise( texCoord.xy ) ) * reflectVec;
        }
    }

    return float4(0.f, 0.f, 0.f, 0.f);
}
```

반사 벡터 계산

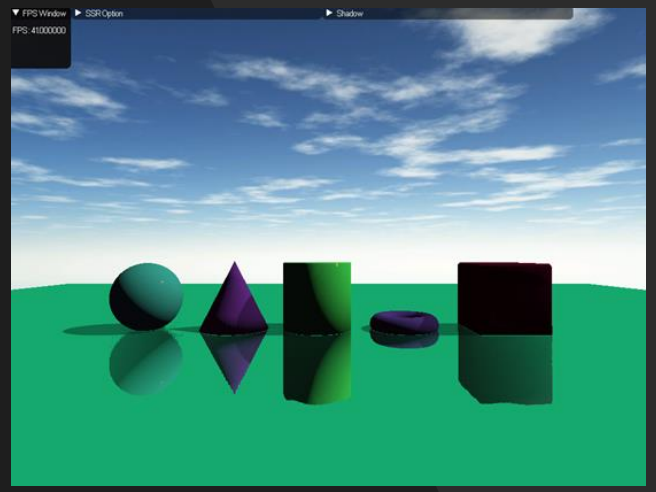
Ray Marching

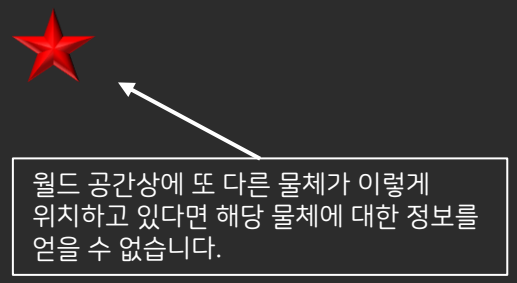
첫 충돌 후 이진 탐색

충돌점을 못 찾았을 때 광선을 진행

Screen Space Reflection Using Ray Marching

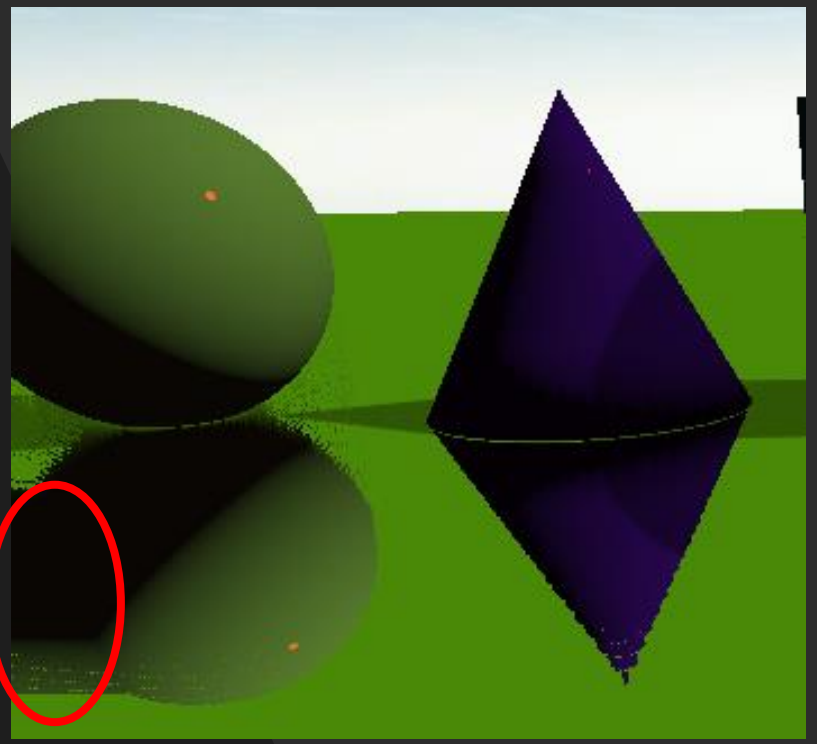
Screen Space Reflection 은 다른 화면 공간 알고리즘 (대표적으로 SSAO)과 마찬가지로 해결할 수 없는 약점을 가지고 있는데 화면 공간 외의 기하정보를 알지 못합니다.



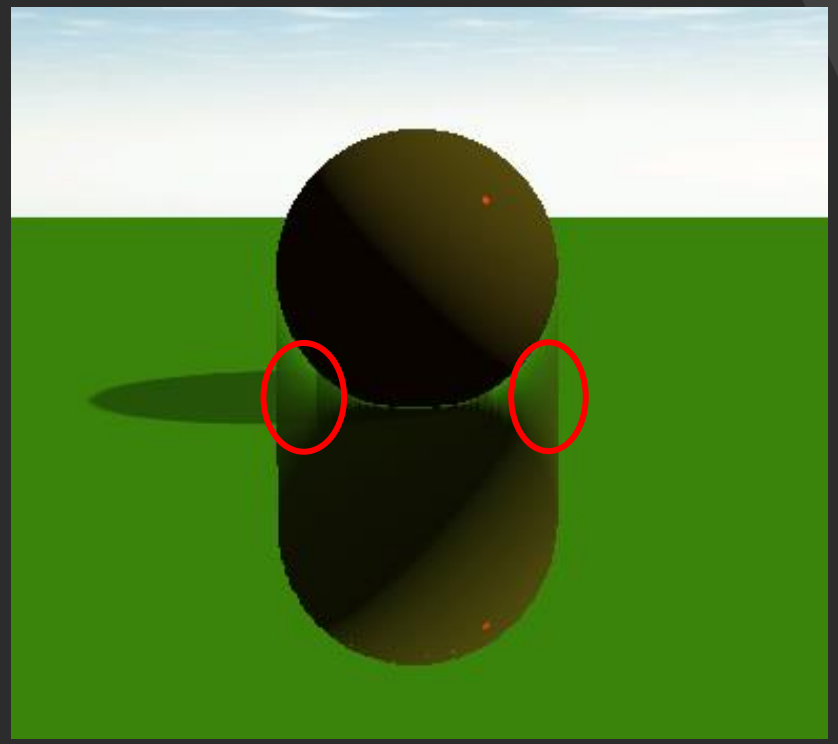


Screen Space Reflection Using Ray Marching

정보를 얻을 수 없는 경우에는 다음과 같은 아티팩트를 확인할 수 있습니다.



윈도우 밖에 있는 경우



물체에 가려진 경우

Screen Space Reflection Using Ray Marching

이런 아티팩트로 인한 어색한 비주얼을 최소화하기 위해서 다음과 같은 처리를 합니다.

1. 윈도우 가장자리에 가까우면 Fade out 시킴

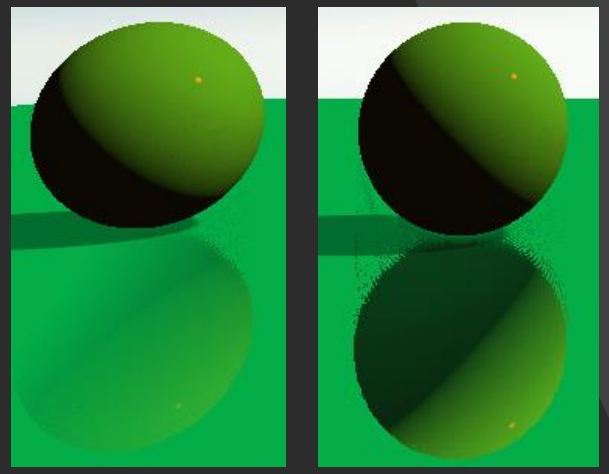
```
float edgeFade = 1.f - pow( length( texCoord.xy - 0.5f ) * 2.f, 2.f );
reflectColor.a *= pow( 0.75f, (length( reflectPos - input.viewPos ) / g_maxRayLength) ) * edgeFade;
return reflectColor;
```

2. 일정 크기의 두께 값 혹은 두께를 저장한 텍스처를 사용

```
float thickness = g_maxThickness / g_FarPlaneDist; // 일정 크기의 두께 값

// ...

if ( depthDiff > g_depthbias && depthDiff < thickness )
{
    float4 reflectColor = BinarySearch( reflectVec, reflectPos );
    // ...
}
```



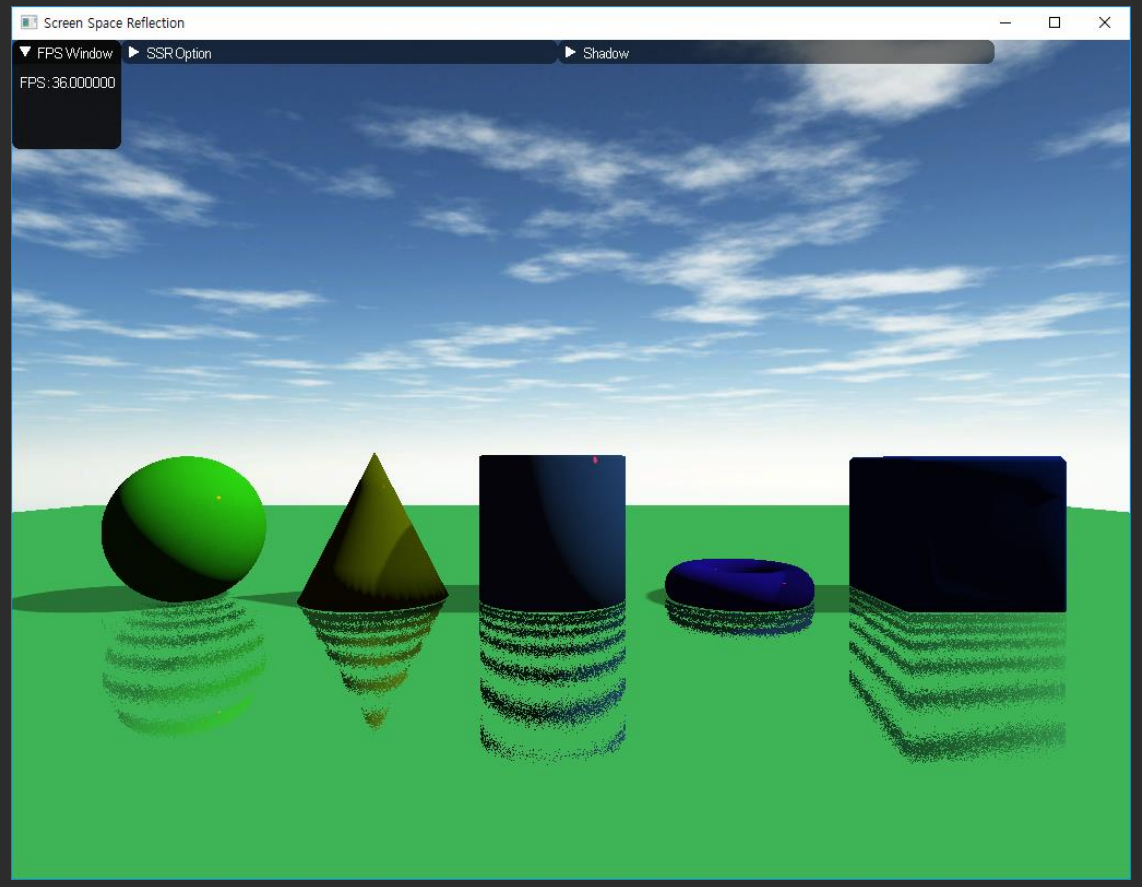
Screen Space Reflection Using Ray Marching

Ray Marching을 사용한 Screen Space Reflection 기법에서 마지막으로 광선의 진행 길이에 대해 살펴보도록 하겠습니다.

광선의 진행 길이는 고정 값을 사용하는 것이 아니라 사용자의 설정 값에 따라 변동 될 수 있습니다.

진행 길이를 늘리면 Ray Marching이 좀 더 빨리 끝나 성능상 이득을 얻을 수 있나 적절하지 않은 진행 길이를 설정하면 다음과 같은 아티팩트를 관찰 할 수 있습니다.

Screen Space Reflection Using Ray Marching



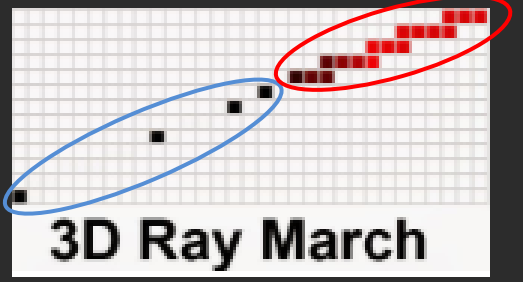
Screen Space Reflection Using Ray Marching

진행 길이가 너무 늘어나면 어떤 위치에서는 광선이 물체를 그냥 통과해버리게 되고 그 결과 반사된 상에 군데 군데 구멍이 생기게 됩니다.

광선의 진행 길이는 장면에 따라 신중하게 결정해야 하는데 이것이 쉽지 않습니다.

3차원 공간에서 Ray Marching은 원근법 때문에 어떤 픽셀은 과하게 샘플링 될 수도 있고 어떤 픽셀은 덜 샘플링 될 수 있습니다.

덜 샘플링 된 부분 광선 경로에 있는 몇몇 픽셀을 건너 뛰는 것이 관찰됨



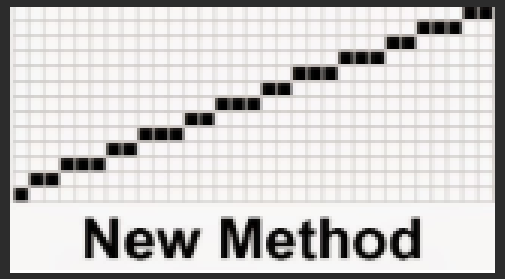
과하게 샘플링 된 부분 같은 픽셀을 여러 번 샘플링 할 수록 붉은색

화면 공간에서 본 3D Ray March

출처 : http://casual-effects.blogspot.com/2014/08/screen-space-ray-tracing.html

Screen Space Ray Tracing

3차원 공간의 Ray Marching의 문제를 해결하기 위해서 화면 공간에서 Ray Marching을 수행합니다.



화면 공간에서의 Ray Marching은 선분 그리기 알고리즘인 DDA(Digital Differential Analyzer) 알고리즘을 사용합니다.

출처 : http://casual-effects.blogspot.com/2014/08/screen-space-ray-tracing.html

Digital Differential Analyzer

DDA 알고리즘은 다음과 같은 단계로 수행됩니다.

1. 그리고자 하는 선분의 기울기를 계산 ($m = \frac{y_{end} - y_{start}}{x_{end} - x_{start}}$)
2. 기울기가 양수이고 $m \leq 1$ 일 경우 x축 방향으로 1식 증가하면서 y값을 계산

$$x_{k+1} = x_k + 1$$

$$y_{k+1} = y_k + m$$

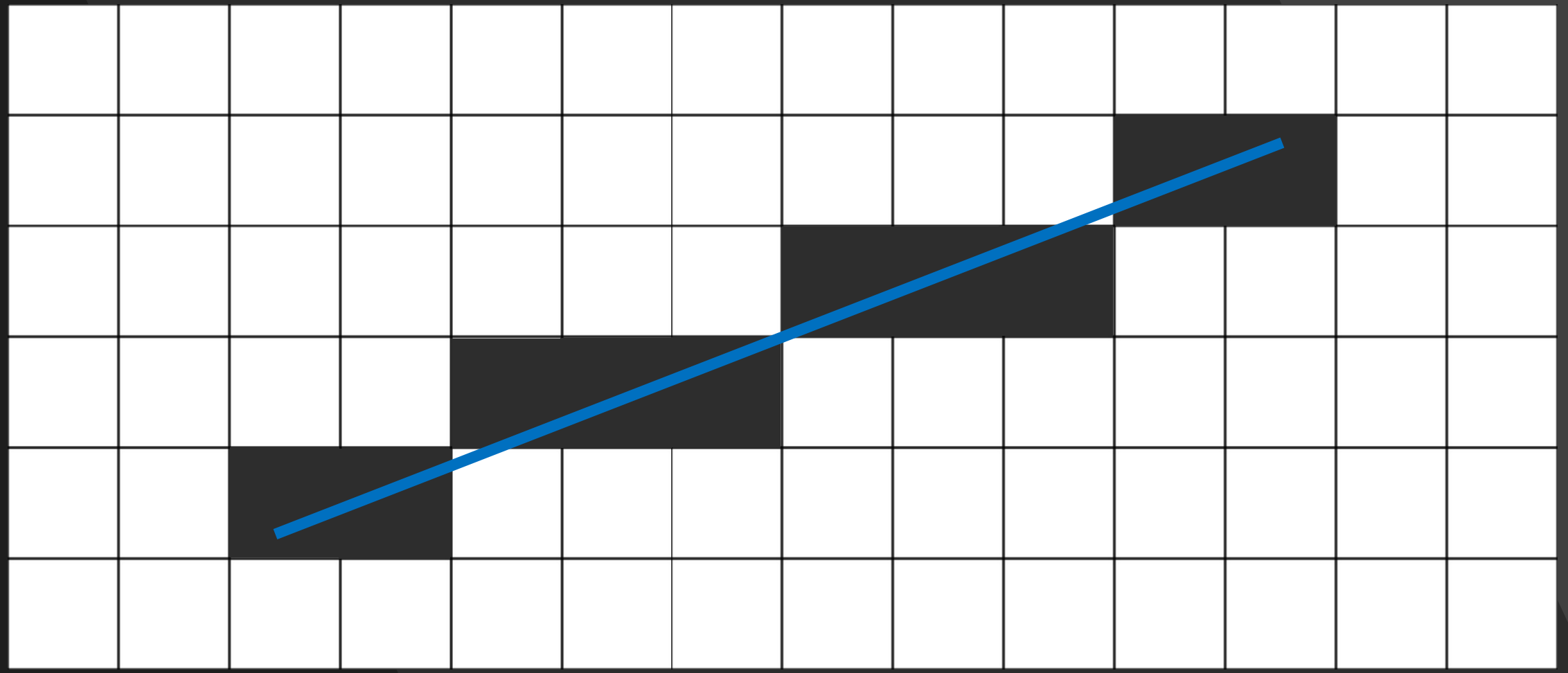
3. $1 < m$ 일 경우 y축 방향으로 1식 증가하면서 x값을 계산

$$x_{k+1} = x_k + \frac{1}{m}$$

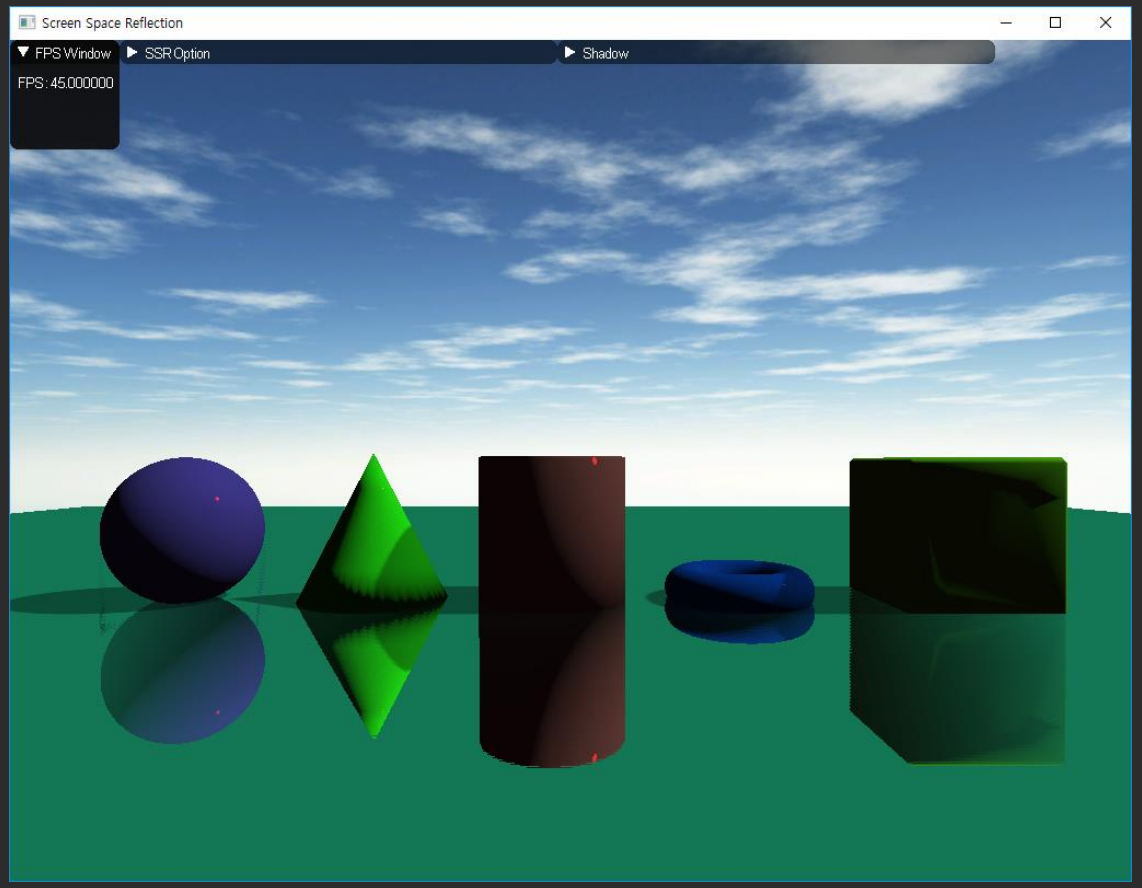
$$y_{k+1} = y_k + 1$$

4. 기울기가 음수 일 경우에도 픽셀 위치를 계산하기 위해 유사한 계산이 이뤄짐

Digital Differential Analyzer



Screen Space Reflection
Using Digital Differential Analyzer



Screen Space Reflection Using Digital Differential Analyzer

```
float4 main( PS_INPUT input ) : SV_TARGET
{
    float3 incidentVec = normalize( input.viewPos );
    float3 viewNormal = normalize( input.viewNormal );

    float3 reflectVec = reflect( incidentVec, viewNormal );
    reflectVec = normalize( reflectVec );
```

반사 벡터 계산

```
    float3 hitPixel_alpha;
    bool isHit = TraceScreenSpaceRay( reflectVec, input.viewPos, hitPixel_alpha );
```

충돌 여부 및 충돌한 픽셀 좌표를 계산

```
    float3 color = framebufferTex.SampleLevel( baseSampler, hitPixel_alpha.xy, 0 ).rgb;

    return float4( color, hitPixel_alpha.z ) * isHit;
}


bool TraceScreenSpaceRay( float3 dir, float3 viewPos, out float3 hitPixel_alpha )
{
    float rayLength = ( viewPos.z + dir.z * g_maxRayLength ) < g_nearPlaneDist ? ( g_nearPlaneDist - viewPos.z ) / dir.z : g_maxRayLength;

    hitPixel_alpha = float3( -1, -1, 1 );

    float3 rayEnd = viewPos + dir * rayLength;

    float4 ssRayBegin = mul( float4( viewPos, 1.0 ), g_projectionMatrix );
    float4 ssRayEnd = mul( float4( rayEnd, 1.0 ), g_projectionMatrix );

    float k0 = 1 / ssRayBegin.w;
    float k1 = 1 / ssRayEnd.w;
```

화면 공간으로 광선의 시작점, 끝점을 변환

Screen Space Reflection Using Digital Differential Analyzer

```
float2 P0 = ssRayBegin.xy * k0;
float2 P1 = ssRayEnd.xy * k1;

P0 = ( P0 * 0.5 + 0.5 ) * g_targetSize;
P1 = ( P1 * 0.5 + 0.5 ) * g_targetSize;
```

화면 공간으로 광선의 시작점, 끝점을 변환

```
P1 += ( dot( P1 - P0, P1 - P0 ) < 0.0001 ) ? float2( 0.01, 0.01 ) : float2( 0, 0 );
float2 delta = P1 - P0;

bool permute = false;
if ( abs( delta.x ) < abs( delta.y ) )
{
    permute = true;
    delta = delta.yx;
    P0 = P0.yx;
    P1 = P1.yx;
}

float stepDir = sign( delta.x );
float invdx = stepDir / delta.x;

float stepCount = 0;
float rayZ = viewPos.z;
float sceneZMax = viewPos.z + 100.f;

float end = stepDir * P1.x;

float3 Pk = float4( P0, k0 );
float3 dPk = float4( float2( stepDir, delta.y * invdx ), ( k1 - k0 ) * invdx );

dPk *= g_rayStepScale;

float thickness = g_maxThickness;
```

기울기 계산 후 DDA delta 값을 계산

$$1 < \frac{y_{end} - y_{start}}{x_{end} - x_{start}}$$
$$= x_{end} - x_{start} < y_{end} - y_{start}$$

Screen Space Reflection Using Digital Differential Analyzer

```
[loop]
for ( ;
    ( ( Pk.x * stepDir ) <= end ) &&  // 광선이 끝까지 진행하기 전까지
    ( stepCount < g_maxRayStep ) &&  // 최대 Ray Marching 단계에 도달하기 전까지
    ( ( rayZ < sceneZMax ) ||
    ( ( rayZ - thickness ) > sceneZMax ) ) &&  // 물체에 부딪히기 전까지
    ( sceneZMax != 0.0 );
    Pk += dPk,
    stepCount += 1 )  // 화면 공간에서 광선을 진행
{
    hitPixel_alpha.xy = permute ? Pk.yx : Pk.xy;
    hitPixel_alpha.xy *= g_invTargetSize;
    hitPixel_alpha.y = 1 - hitPixel_alpha.y;

    rayZ = 1 / Pk.w;
    sceneZMax = depthbufferTex.SampleLevel( baseSampler, hitPixel_alpha.xy, 0 ).x;
    thickness = backfaceDepthBufferTex.SampleLevel( baseSampler, hitPixel_alpha.xy, 0 ).x * g_FarPlaneDist;
    sceneZMax *= g_FarPlaneDist;
    thickness -= sceneZMax;
    sceneZMax += g_depthbias;
}
```

Ray Marching

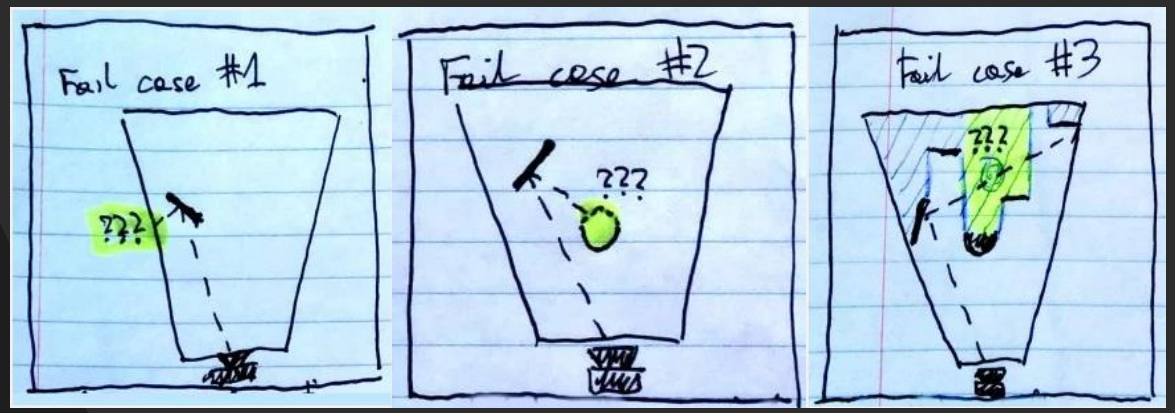
```
    float edgeFade = 1.f - pow( length( hitPixel_alpha.xy - 0.5 ) * 2.f, 3.0f );
    hitPixel_alpha.z = edgeFade;
    return ( rayZ >= sceneZMax ) && ( rayZ - thickness <= sceneZMax );
}
```

Summary

기존 반사 표현 방식의 단점을 보완할 수 있는 대안 기술이라고 할 수 있습니다.

성능이 장면의 복잡도에 영향을 받지 않지만 비용이 낮다고는 할 수 없습니다.

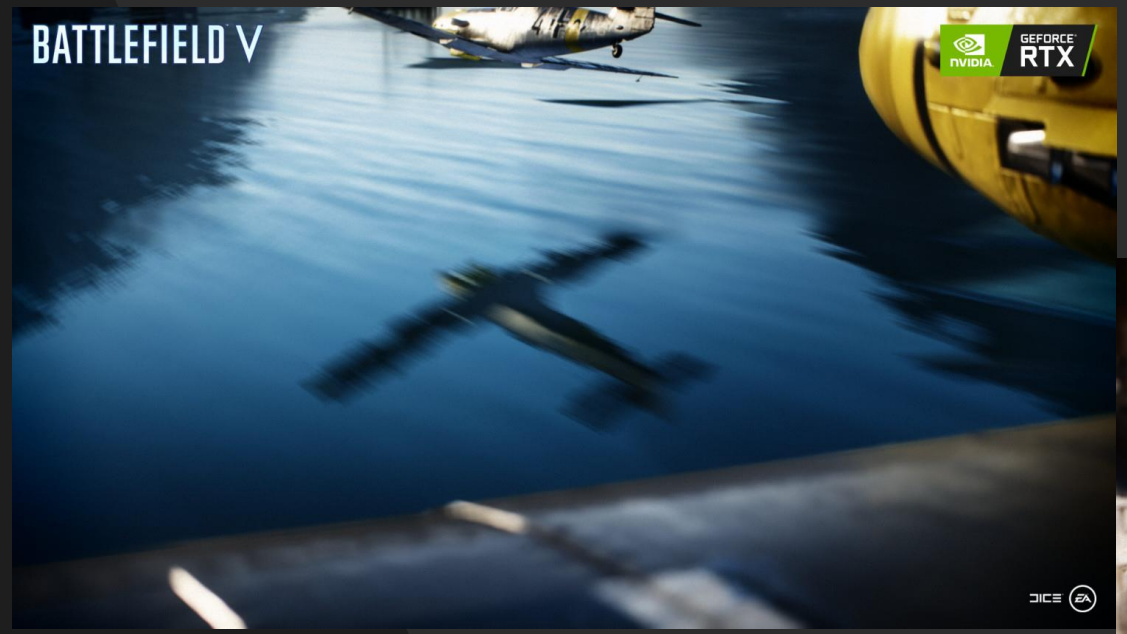
2차원 텍스처로 기록된 장면 정보만을 사용하기 때문에 정보 부족으로 인한 다양한 문제가 발생 할 수 있습니다.



왼쪽부터 차례대로
1. 절두체 밖의 정보 X
2. 물체 뒷부분의 정보 X
3. 가려진 물체에 대한 정보 X

출처 : http://casual-effects.blogspot.com/2014/08/screen-space-ray-tracing.html

Future?



출처 : https://www.techpowerup.com/247365/dice-to-dial-back-ray-tracing-eye-candy-in-battlefield-v-to-favor-performance

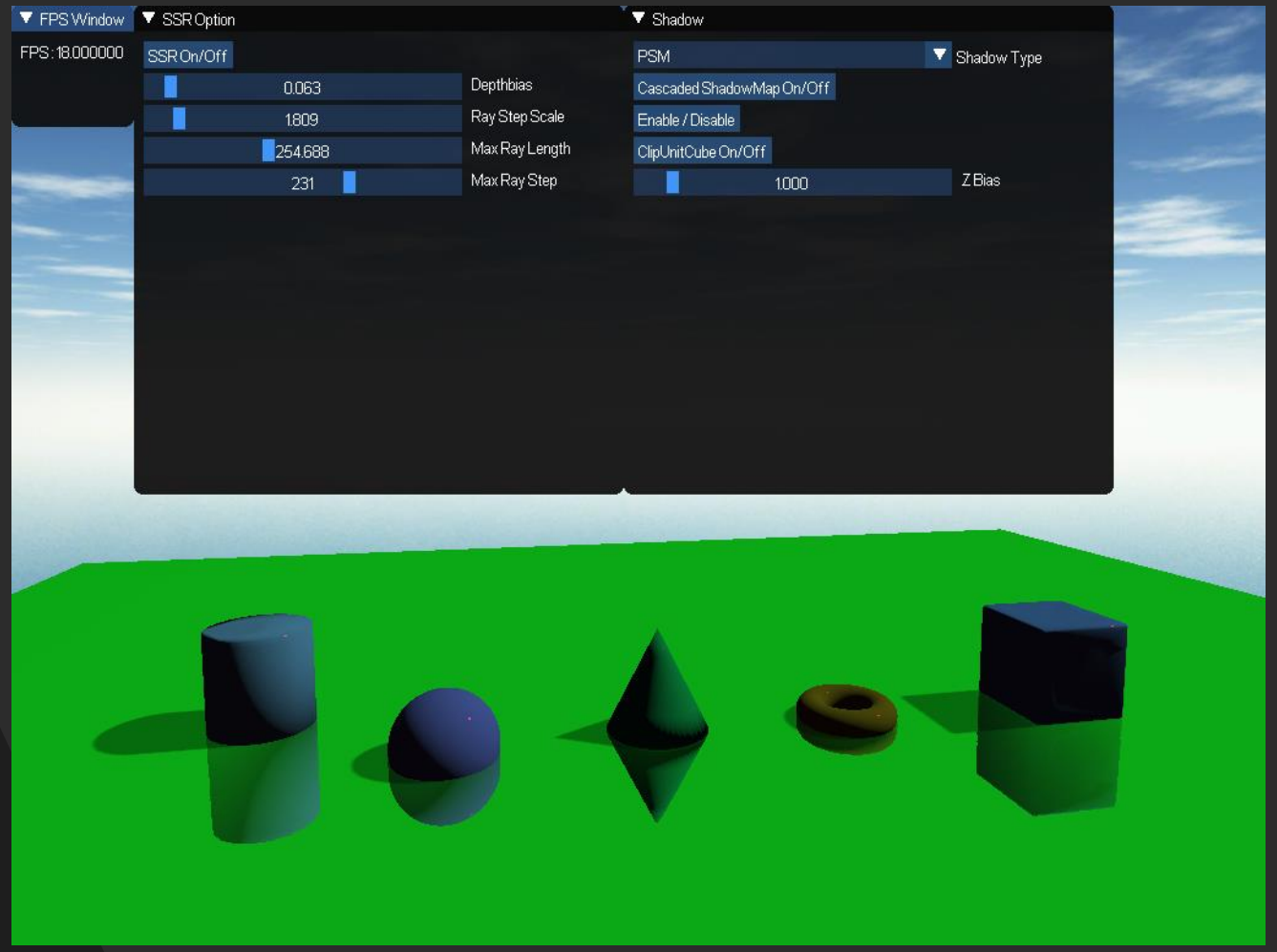
Future?

2018년도 9월 20일에 출시한 GeForce RTX 20 시리즈에서 실시간 레이 트레이싱 기능을 지원합니다.

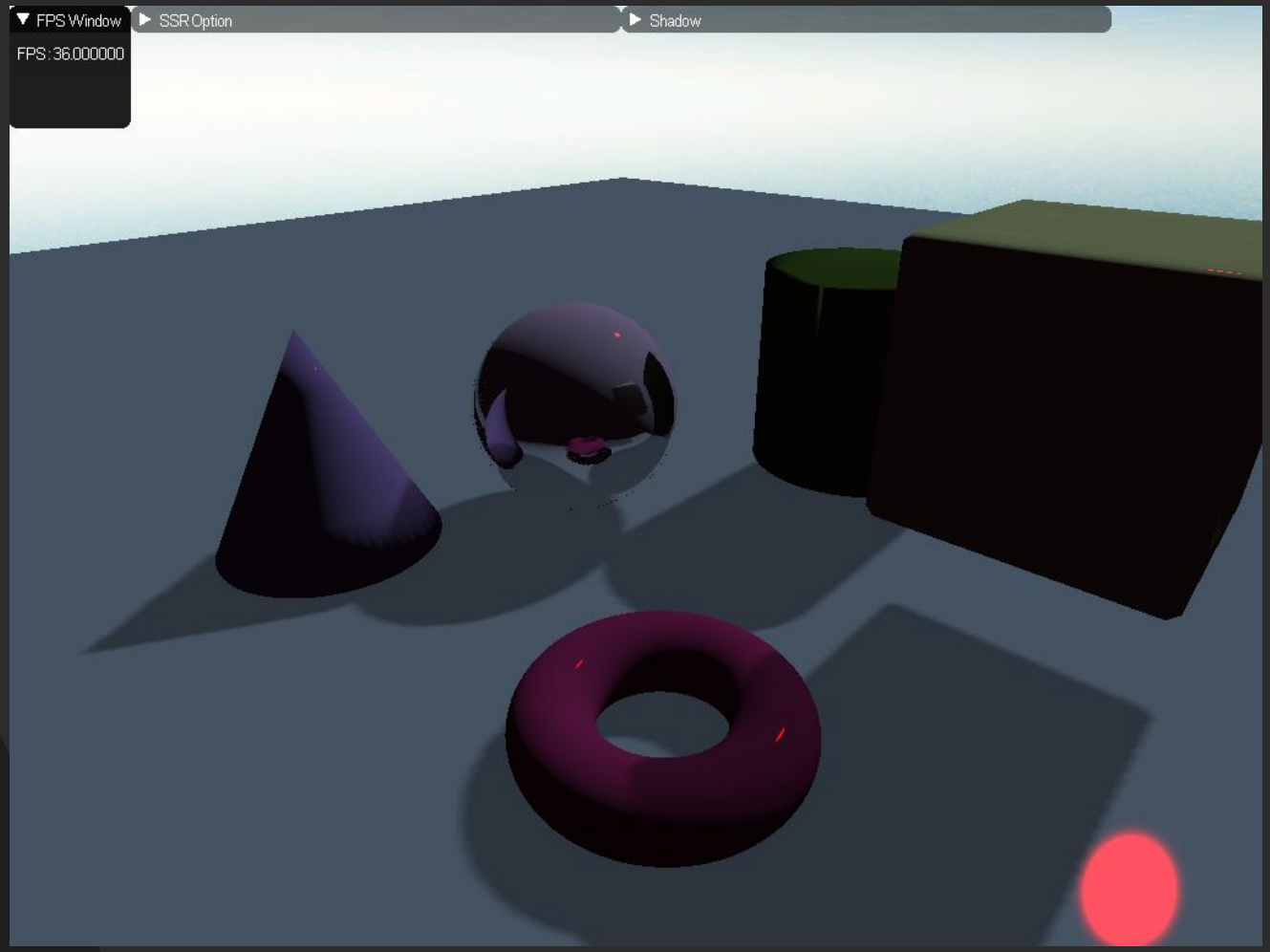
실시간 레이 트레이싱을 통해서 기존 기술들의 한계점에서 벗어나 반사 뿐만이 아니라 굴절, 그림자도 계산할 수 있습니다.

다만 실시간 레이 트레이싱이 하드웨어에 종속적인 기술이고 가격대가 높게 책정되어 있어 (RTX 2070이 60만원 후반) 보편적으로 정착할 수 있을지 두고 볼 필요가 있습니다.

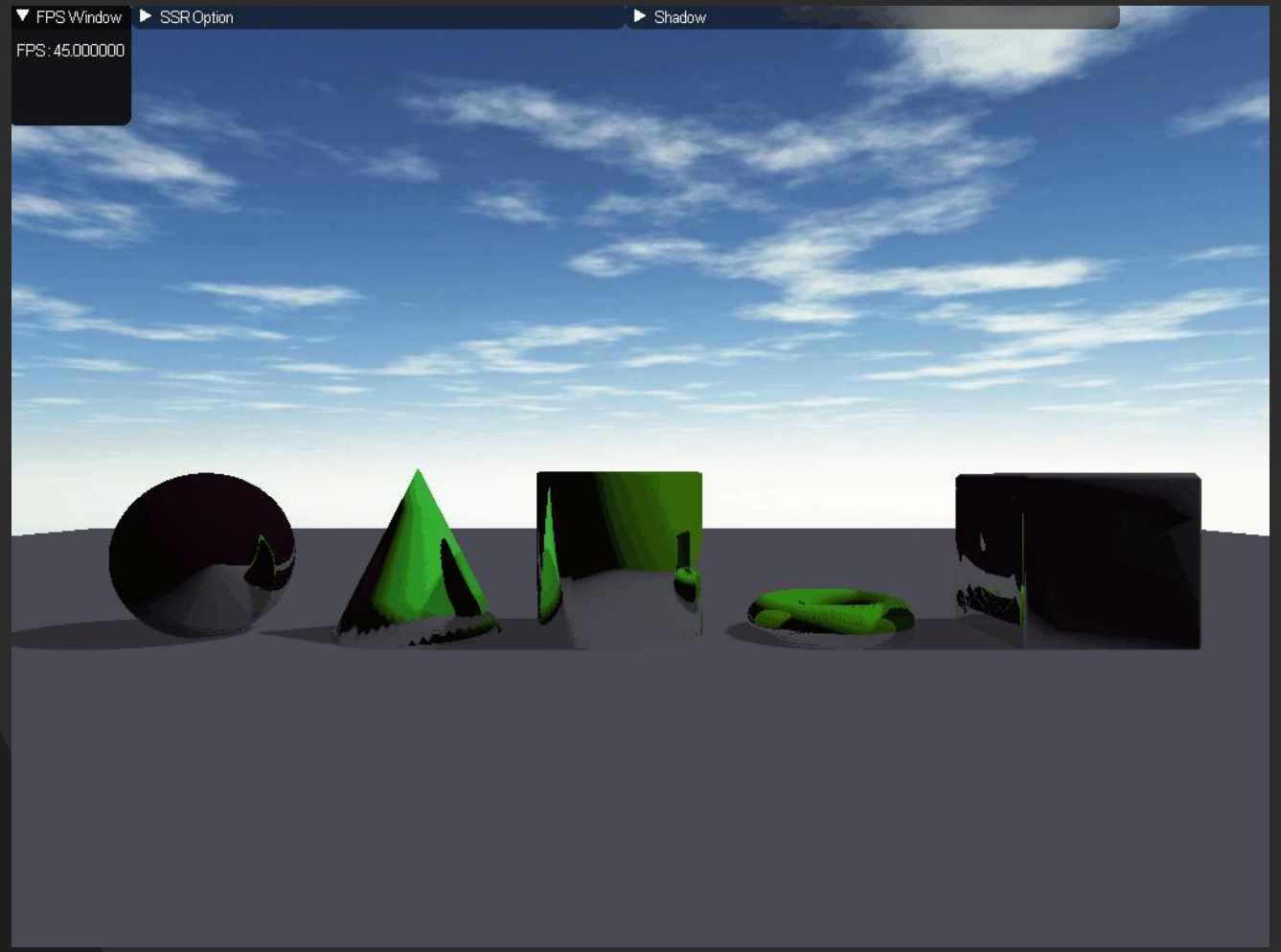
Screenshot


FPS Window
FPS : 18.000000
SSR Option
SSR On/Off
0.063
Depthbias
1.809
Ray Step Scale
254.688
Max Ray Length
231
Max Ray Step
Shadow
PSM
Shadow Type
Cascaded ShadowMap On/Off
Enable / Disable
ClipUnitCube On/Off
1.000
Z Bias

Screenshot



Screenshot



Reference

아이콘 출처 : https://www.flaticon.com/

구와 광선의 충돌 : Real-Time Collision Detection (Christer Ericson)

Screen Space Ray Tracing : http://casual-effects.blogspot.com/2014/08/screen-space-ray-tracing.html

Battlefield V: Official GeForce RTX Real-Time Ray Tracing Demo : https://youtu.be/WoQr0k2IA9A

그 밖에 좋은 사이트 들 :

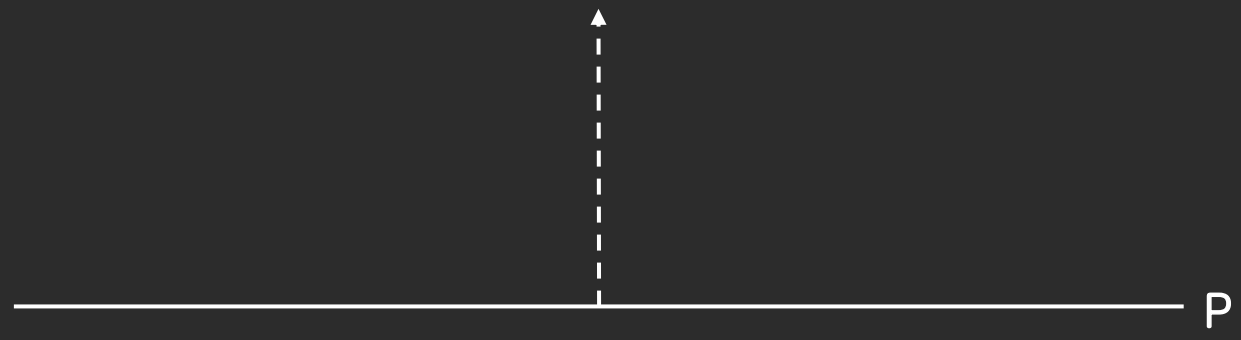
http://www.kode80.com/blog/2015/03/11/screen-space-reflections-in-unity-5/

http://tips.hecomi.com/entry/2016/04/04/022550

Appendix

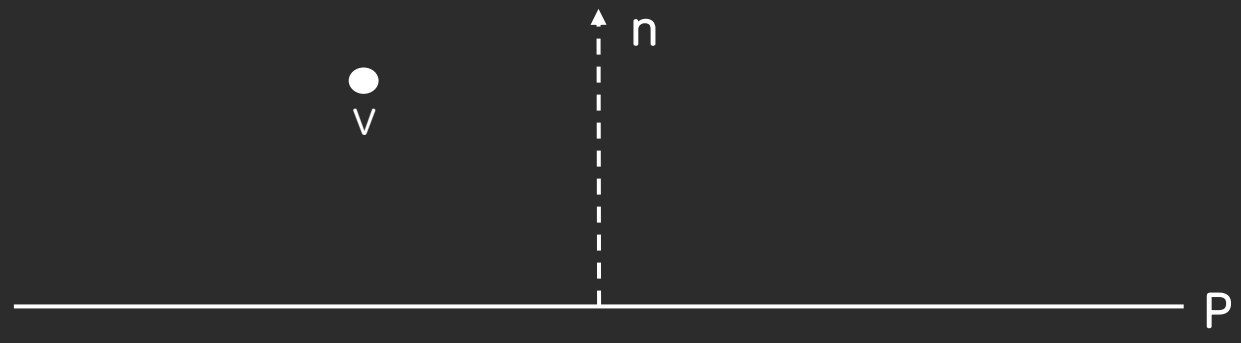
Reflection Matrix

거울에 맺히는 상의 위치를 찾기 위해서 임의의 평면 P를 생각해 보겠습니다.



Reflection Matrix

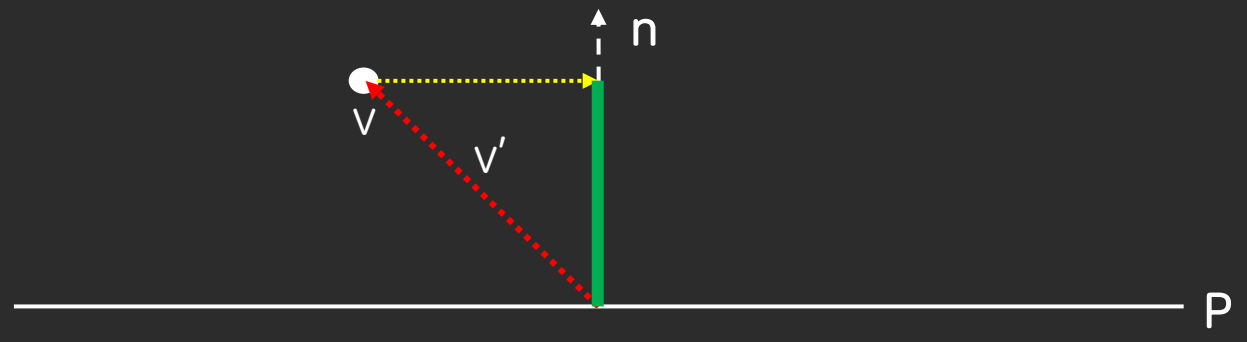
그리고 한 점 v가 다음과 같이 위치해 있다고 하겠습니다.



Reflection Matrix

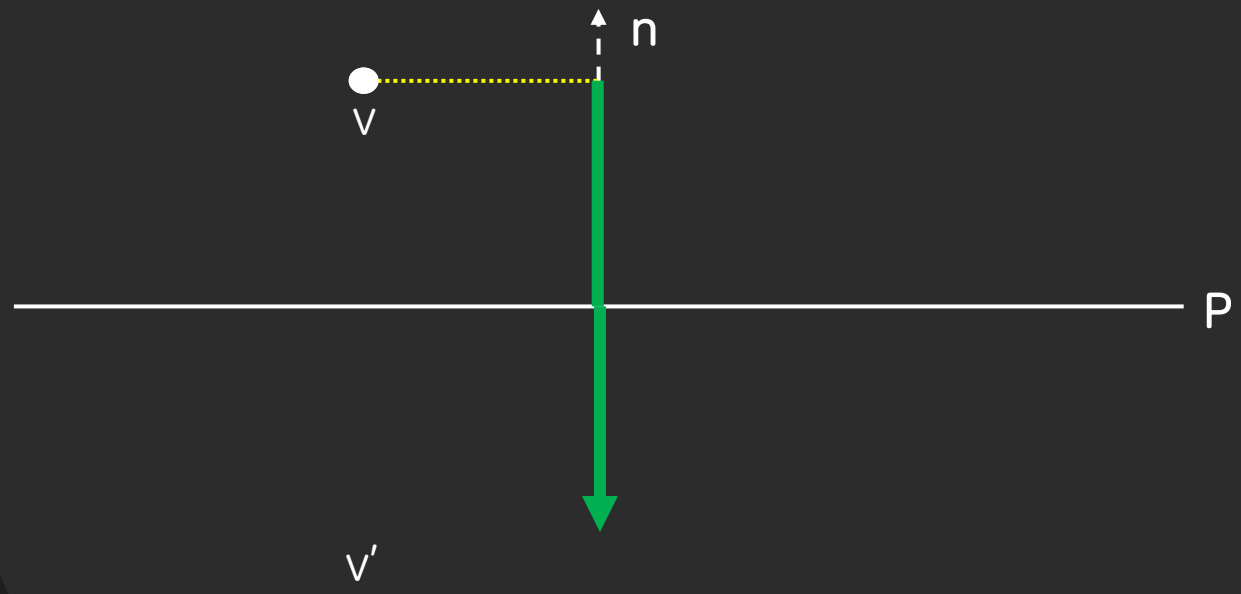
임의의 평면 P를 기준으로 v가 대칭 이동한 위치를 구해야 합니다.

1. 점 v의 위치를 벡터로 취급하여 평면 P의 법선 벡터와 내적합니다. 그 결과로 n에 투영된 v의 길이를 알 수 있습니다.



Reflection Matrix

2. 평면 P에 대해서 대칭 이동한 v의 위치는 이 길이($v' \cdot n$) 의 2배 만큼의 거리를 법선 n의 반대 방향으로 이동한 위치가 됩니다.



Reflection Matrix

수식으로 정리하자면

$$v - 2 * (v' \cdot n) * n$$

가 되고 이를 행렬로 표현하면 다음과 같습니다.

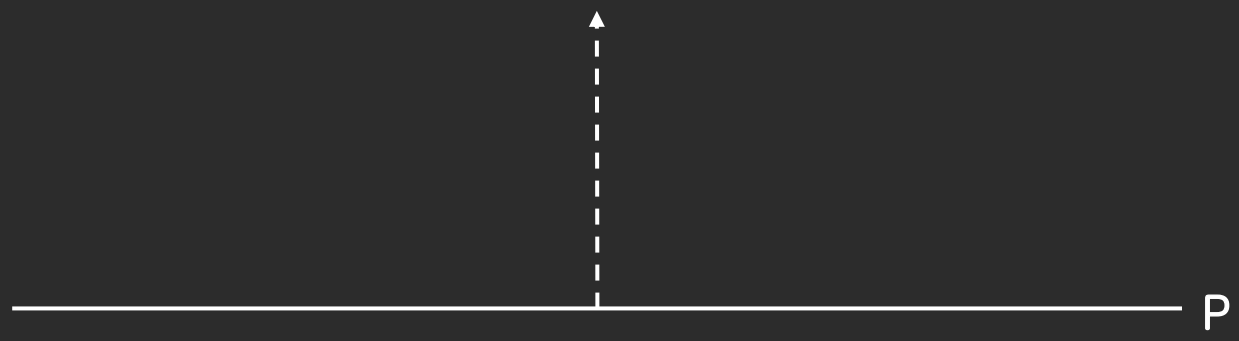
$$I - 2P^T N$$

$$= \begin{pmatrix} 1 - 2{n_x}^2 & -2n_x n_y & -2n_x n_z & 0 \\ -2n_x n_y & 1 - 2{n_y}^2 & -2n_z & 0 \\ -2n_x n_y & -2n_y n_z & 1 - 2{n_z}^2 & 0 \\ -2dn_x & -2d & -2dn_z & 1 \end{pmatrix}$$

Reflection Vector

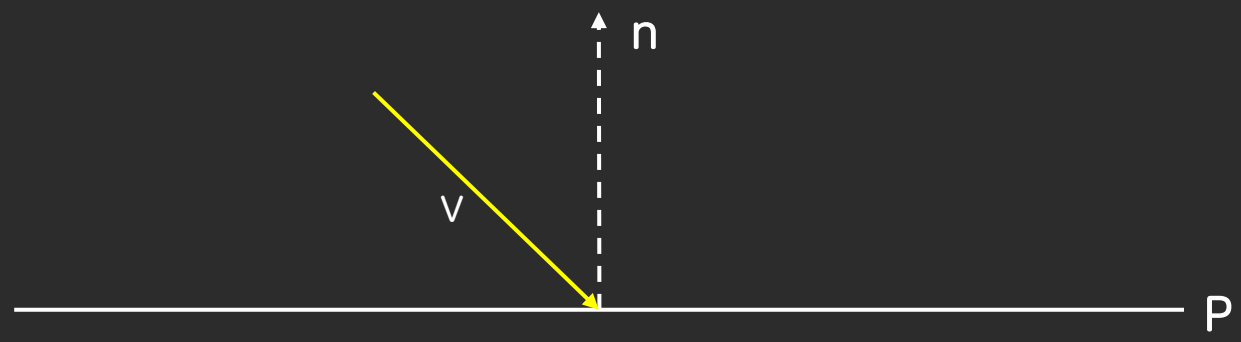
반사 벡터를 텍스처 좌표로 바로 사용할 수 있습니다. Cube 텍스처만 있다면 반사 벡터를 구하는 방법만 알면 됩니다.

다시 임의의 평면 P를 생각해 봅시다.



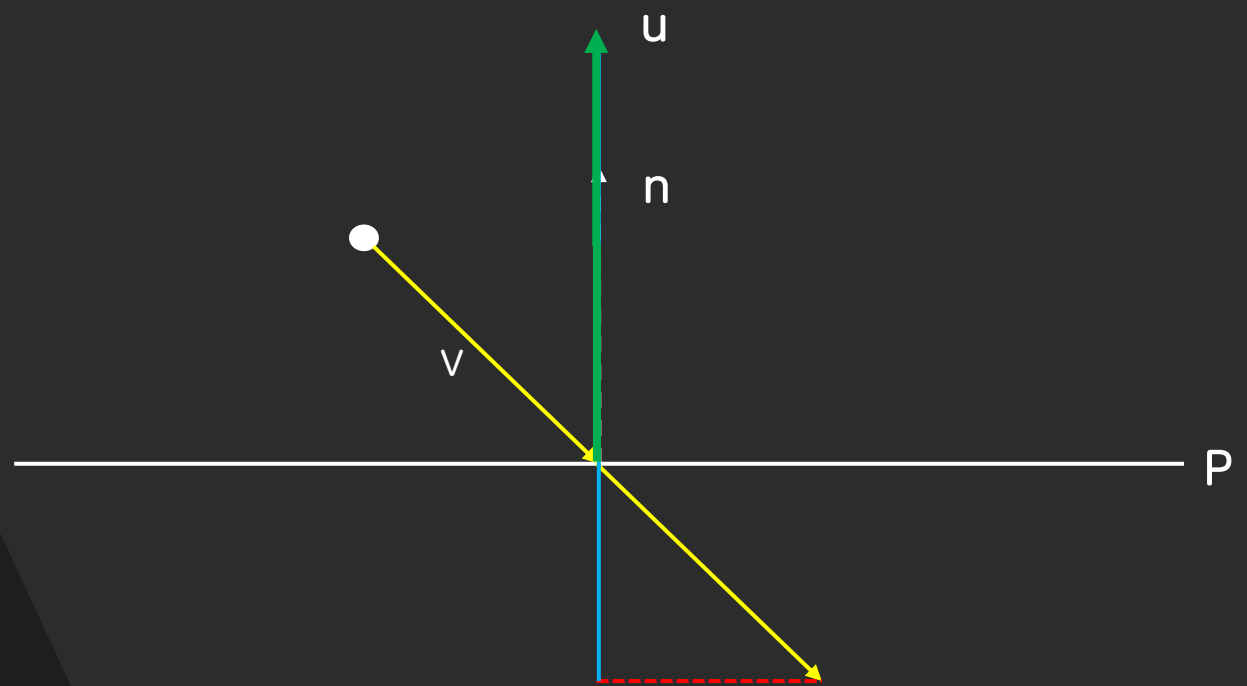
Reflection Vector

그리고 다음과 같은 입사 벡터 v가 있다고 생각해 봅시다.



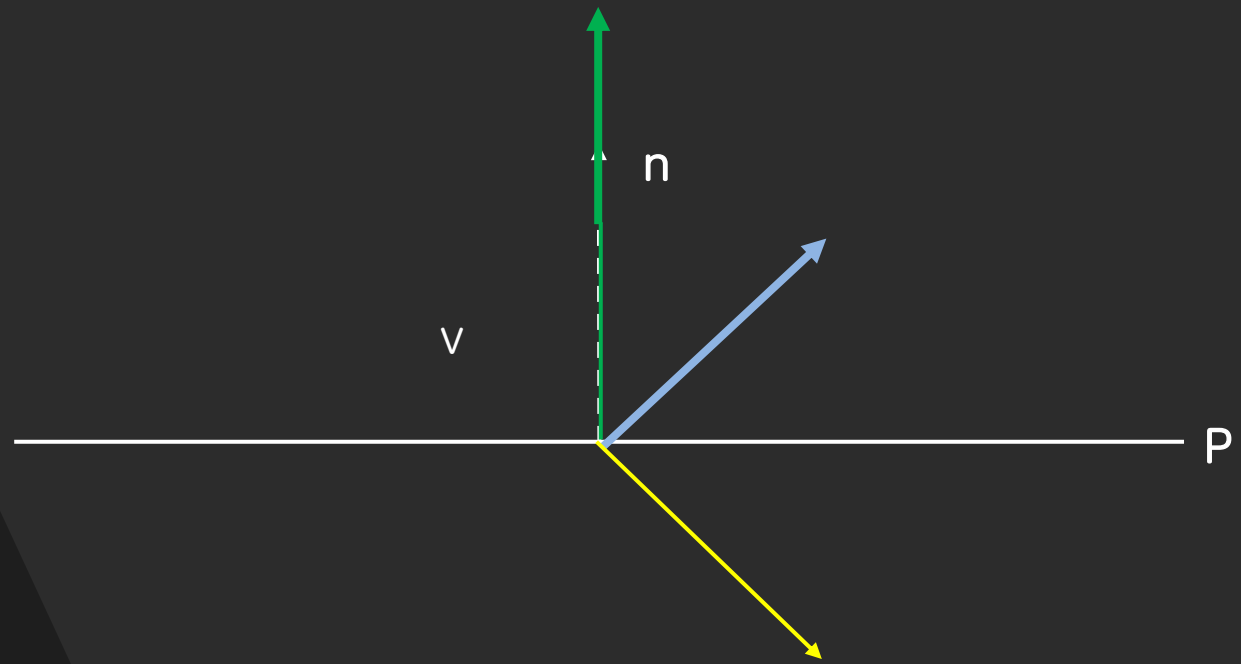
Reflection Vector

1. 이 벡터 v와 평면의 법선 벡터 n을 내적하면 법선 벡터 n에 투영된 v의 길이를 얻을 수 있습니다. 이 값에 -2배를 하여 법선 벡터에 곱하면 다음과 같은 벡터 u를 얻게 됩니다.



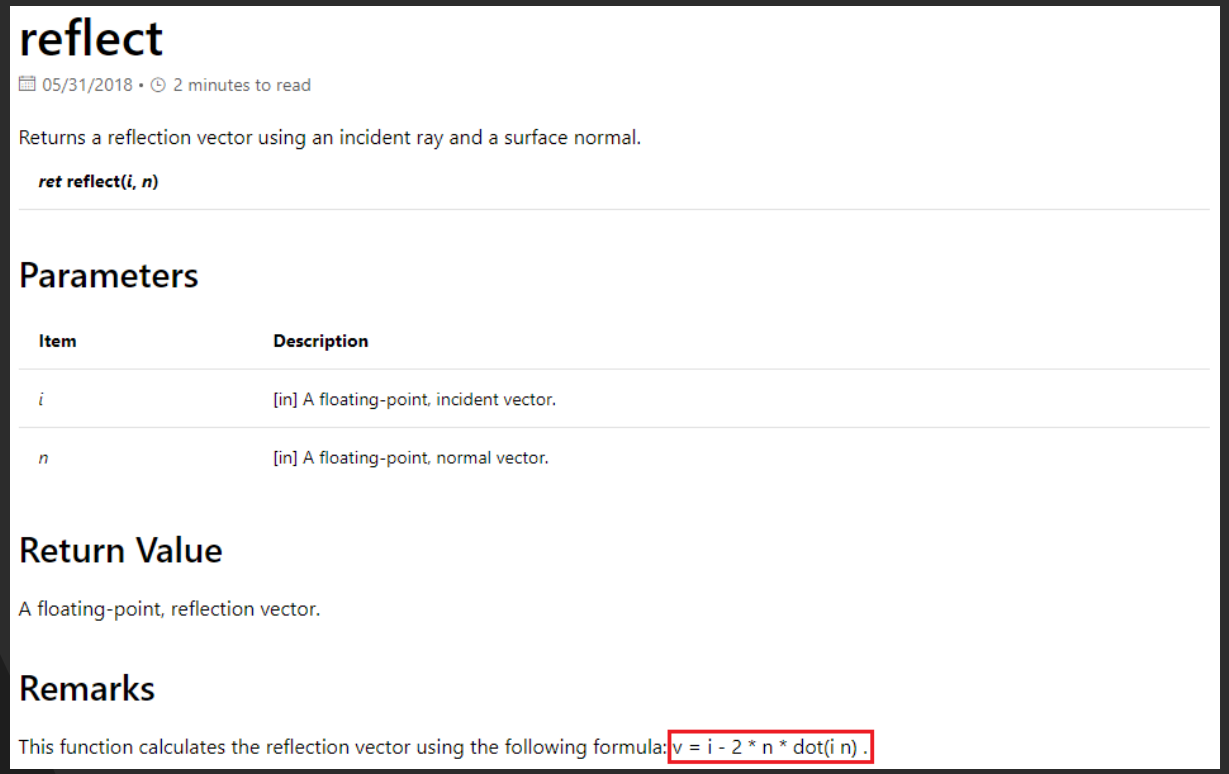
Reflection Vector

2. 이제 이렇게 얻은 벡터 u와 벡터 v를 더하면 반사 벡터를 얻을 수 있습니다.



Reflection Vector

수식으로 정리하면 결국 $v - 2 * n * (v \cdot n)$ 가 되고 hlsl에서는 reflect 함수로 쉽게 구할 수 있습니다.



reflect

05/31/2018 • 2 minutes to read

Returns a reflection vector using an incident ray and a surface normal.

***ret* reflect(*i*, *n*)**

Parameters

Item	Description
i	[in] A floating-point, incident vector.
n	[in] A floating-point, normal vector.

Return Value

A floating-point, reflection vector.

Remarks

This function calculates the reflection vector using the following formula: v = i - 2 * n * dot(i n) .